go.mod 中文文档

go.mod 中文文档，翻译自官方英文文档 https://go.dev/ref/mod，你可以在 https://github.com/JohnGuoy/go.mod-zh-Hans 获取本文档的更新。

目录

Modules 是 Go 管理依赖的方式。

本文档是 Go 模块系统的详细参考手册。有关创建 Go 项目的介绍，请参阅如何编写 Go 代码。有关使用模块、将项目迁移到模块以及其他主题的信息，请参阅从使用 Go 模块开始的博客系列。

模块、包和版本号(Modules, packages, and versions)

模块(module)是一起发布、版本控制和分发的包的集合。模块可以直接从版本控制存储库或模块代理服务器下载。

模块由模块路径(module path)标识，该路径在 go.mod 文件中声明，以及有关模块依赖项的信息。模块根目录(module root directory)是包含 go.mod 文件的目录。主模块(main module)是包含调用 go 命令的目录的模块。

模块中的每个包(package)都是同一目录中一起编译的源文件的集合。包路径(package path)是与包含包的子目录路径（相对于模块根目录）连接在一起的模块路径。 例如，模块“golang.org/x/net”在子目录“html”中包含一个包。该包的路径是“golang.org/x/net/html”。

模块路径(Module paths)

模块路径(module path)是模块的规范名称(canonical name)，在模块的 go.mod 文件中使用模块指令(module directive)声明。模块的路径是模块内包路径的前缀。

模块路径应该描述模块的作用以及在哪里可以找到它。通常，模块路径由存储库根路径、存储库中的目录（通常为空）和主要版本后缀（仅适用于主版本号 2 或更高版本号）组成。

- 存储库根路径(repository root path)是模块路径的一部分，对应于开发模块的版本控制存储库的根目录。大多数模块都在其存储库的根目录中定义，因此这通常就是模块的整个路径。例如，golang.org/x/net 是同名模块的存储库根路径。有关 go 命令如何使用模块路径对应的 HTTP 请求定位存储库的信息，请参阅根据模块路径查找对应的存储库。
- 如果模块未在存储库的根目录中定义，则模块子目录(module subdirectory)是模块路径的一部分，不包括主版本号后缀。 这也用作语义版本标签(semantic version tags)的前缀。 例如， 模块 golang.org/x/tools/gopls 位于根路径为

golang.org/x/tools 的存储库的 gopls 子目录中，因此它具有模块子目录 gopls。请参阅将包的版本映射到提交(commits)和储存库里面的模块目录。

- 如果模块以主版本号 2 或更高版本号发布，则模块路径必须以主版本号后缀（如/v2）结尾。这可能是也可能不是子目录名称的一部分。例如，路径为 golang.org/x/repo/sub/v2 的模块可能位于存储库 golang.org/x/repo 的/sub 或/sub/v2 子目录中。

如果一个模块可能被其他模块依赖，则必须遵循这些规则，以便 go 命令可以找到并下载该模块。模块路径中允许的字符也有一些词法限制。

版本号(Versions)

版本号标识模块的不可变快照，它可以是发布版或预发布版。每个版本号都以字母 v 开头，然后是语义版本号。有关如何格式化、解释和比较版本号的详细信息，请参阅语义版本号控制 2.0.0。

总而言之，语义版本号由三个由点分隔的非负整数（主版本号、次版本号和补丁版本号，从左到右）组成。补丁版本号后面可以跟一个以连字符开头的可选预发布字符串(pre-release string)。预发布字符串或补丁版本号后面可以跟以加号开头的构建元数据字符串(build metadata string)。例如，v0.0.0、v1.12.134、v8.0.5-pre 和 v2.0.9+meta 是有效版本号。

版本号的每个部分都表示该版本是否稳定以及是否与以前的版本兼容。

- 在对模块的公开接口或文档记录的功能进行向后不兼容的更改后（例如，在删除软件包后），必须增加主版本号，并且必须将次版本号和补丁版本号设置为零。
- 在向后兼容的更改之后，例如，在添加新功能之后，必须增加次版本号并将补丁版本号设置为零。
- 补丁版本号必须在不影响模块公开接口的更改后递增，例如 bug 修复或优化。
- 预发布后缀表示版本是预发布版本。预发布版本号排在相应的发布版本号之前。例如，v1.2.3-pre 出现在 v1.2.3 之前。
- 比较版本号的时候会忽略构建元数据后缀(build metadata suffix)。带有构建元数据的标签(tags)在版本控制存储库中会被忽略，但构建元数据保留在 go.mod 文件中指定的版本号中。后缀+incompatible 表示在迁移到模块版本主版本号 2 或更高版本号之前发布的版本（请参阅与非模块存储库的兼容性）。

如果一个版本的主版本号为 0 或具有预发布后缀，则该版本被认为是不稳定的。不稳定的版本不受兼容性要求的约束。例如，v0.2.0 可能与 v0.1.0 不兼容，v1.5.0-beta 可能与 v1.5.0 不兼容。

Go 可以使用不遵循这些约定的标签、分支或修订来访问版本控制系统中的模块。但是，在主模块中，go 命令会自动将不遵循此标准的修订名称转换为规范版本号。作为此过程的一部分，go 命令还将删除构建元数据后缀（+incompatible 除外）。 这可能会产生一个伪版本号，一个预发布版本号，它对修订标识符（例如 Git 提交哈希(commit hash)）和来自版本控制系统的时间戳进行编码。例如，命令 go get -d golang.org/x/net@daa7c041 会将提交哈希 daa7c041 转换为伪版本号

v0.0.0-20191109021931-daa7c04131f5。在主模块之外需要使用规范版本号，如果 go.mod 文件中出现 master 等非规范版本号，go 命令会报错。

伪版本号(Pseudo-versions)

伪版本号(pseudo-version)是一种特殊格式的预发布版本号，它对版本控制存储库中特定修订的信息进行编码。例如，v0.0.0-20191109021931-daa7c04131f5 是一个伪版本。

伪版本号可以指没有语义版本号标签可用的修订号。它们可用于在创建版本号标签之前测试提交(test commits)，例如，在开发分支上。

每个伪版本号包含三个部分：

- 一个基础版本号(base version)前缀（vX.0.0 或 vX.Y.Z-0），它要么派生自修订版号之前的语义版本号标签，要么派生自 vX.0.0（如果没有此类标签）。
- 时间戳（yyyymmddhhmmss），它是创建修订的 UTC 时间。在 Git 中，这是提交时间(commit time)，而不是作者时间(author time)。
- 一个修订标识符（abcdefabcdef），它是提交哈希的 12 个字符的前缀，或者在 Subversion 版本控制系统中，一个以 0 填充的修订号。

每个伪版本号可能是三种形式之一，具体取决于基础版本号。这些形式确保伪版本号比较起来高于其基础版本号，但低于下一个版本号。

- 当没有已知的基础版本号时，使用 vX.0.0-yyyymmddhhmmss-abcdefabcdef。与所有版本号一样，主版本号 X 必须与模块的主版本号后缀匹配。
- vX.Y.Z-pre.0.yyyymmddhhmmss-abcdefabcdef 用于当基础版本号是像 vX.Y.Z-pre 这样的预发布版本号时。
- vX.Y.(Z+1)-0.yyyymmddhhmmss-abcdefabcdef 用于基础版本号是像 vX.Y.Z 这样的发布版本号。例如，如果基础版本号是 v1.2.3，则伪版本号可能是 v1.2.4-0.20191109021931-daa7c04131f5。

通过使用不同的基础版本号，多个伪版本号可能会指向同一个提交(commit)。当在编写伪版本号后给较低版本号打标签(tag)时，这自然会发生。

这些形式为伪版本号提供了两个有用的属性：

- 具有已知基础版本号的伪版本号排序高于这些版本号，但低于其他更高版本的预发行版的版本号。
- 具有相同基础版本号前缀的伪版本号按时间顺序排序。

go 命令执行多项检查以确保模块作者已掌控如何将伪版本号与其他版本号进行比较，并且伪版本号指向的实际上是模块提交历史(commit history)信息的部分修订。

- 如果指定了基础版本号，则必须有相应的语义版本号标签(tag)，该标签是伪版本号描述的修订的祖先版本。这可以防止开发人员使用比所有打了标签的版本号（如 v1.999.999-99999999999999-daa7c04131f5）更高的伪版本号绕过

最小版本号选择。

- 时间戳必须与修订的时间戳匹配。这可以防止攻击者使用无限数量的其他相同的伪版本号淹没模块代理。这也可以防止模块使用者更改版本号的相对顺序。
- 修订标识符必须是模块存储库的分支或标签之一的祖先。这可以防止攻击者引用未经批准的更改或拉取请求。

伪版本号永远不需要手动输入。许多命令接受提交哈希或分支名称，并将其自动转换为伪版本号（或打了标签的版本号，如果可用）。 例如：

```
go get -d example.com/mod@master
go list -m -json example.com/mod@abcd1234
```

主版本号后缀(Major version suffixes)

从主版本号 2 开始，模块路径必须具有与主版本号匹配的主版本号后缀(major version suffix)，例如/v2。例如，如果一个模块在 v1.0.0 具有路径 example.com/mod，则它必须在 v2.0.0 版本具有路径 example.com/mod/v2。

主版本号后缀实现导入兼容性规则：如果旧包和新包具有相同的导入路径，则新包必须向后兼容旧包。

根据定义，模块的新主版本号中的包不向后兼容以前主版本号中的相应包。因此，从 v2 开始，包需要新的导入路径。这是通过向模块路径添加主版本号后缀来实现的。由于模块路径是模块中每个包的导入路径的前缀，因此将主版本号后缀添加到模块路径可为每个不兼容版本提供不同的导入路径。

主版本号 v0 或 v1 中不允许使用主版本号后缀。v0 和 v1 之间不需要更改模块路径，因为 v0 版本不稳定，没有兼容性保证。此外，对于大多数模块，v1 向后兼容最新的 v0 版本；v1 版本号作为对兼容性的承诺，不是表示与 v0 版本相比起来有不兼容的更改。

作为一种特殊情况，以 gopkg.in/开头的模块路径必须始终具有主版本号后缀，即使 v0 和 v1 版本也是如此。后缀必须以点而不是斜杠开头（例如，gopkg.in/yaml.v2）。

主版本号后缀让一个模块的多个主版本在同一个构建(build)中共存。由于钻石依赖性问题，这可能是必要的。通常，如果依赖关系指向一个模块的两个不同版本，则将使用更高的那个版本。但是，如果这两个版本是不兼容的，那么其中任何一个版本都不可能满足所有客户的需求。由于不兼容的版本必须具有不同的主版本号，因此根据主版本号后缀的规则，它们也一定具有不同的模块路径。这解决了冲突：具有不同后缀的模块被视为各自独立的模块，并且它们的包——即使是相对于它们的模块根路径，位于同一子目录中的包——也是独一无二的。

许多 Go 项目在迁移到模块之前（甚至可能在介绍模块之前）发布了 v2 或更高版本的版本，而不使用主版本号后缀。这些版本使用+incompatible 构建标签(build

tag)进行注释（例如，v2.0.0+incompatible）。有关更多信息，请参阅与非模块存储库的兼容性。

将包解析为模块(Resolving a package to a module)

go 命令在使用包路径(package path)加载包时，需要确定是哪个模块提供了该包。

go 命令首先在构建列表(build list)中搜索具有包路径前缀的路径的模块。例如，如果导入了包 example.com/a/b，并且模块 example.com/a 在构建列表中，则 go 命令将检查 example.com/a 是否在目录 b 中包含包。目录中必须至少存在一个扩展名为.go 的文件，才能将其视为一个包。构建约束(Build constraints)不适用于此目的。如果构建列表中只有一个模块提供这个包，则使用该模块。如果没有模块提供这个包，或者有两个或多个模块提供这个包，go 命令会报错。-mod=mod 标志指示 go 命令尝试去查找提供缺失包的新模块并更新 go.mod 和 go.sum。go get 和 go mod tidy 命令会自动执行此操作。

当 go 命令从包路径查找新模块时，它会检查 GOPROXY 环境变量，这是一个逗号分隔的代理 URL 列表或关键字 direct 或 off。代理 URL 指示 go 命令应该使用 GOPROXY 协议(GOPROXY protocol)联系模块代理(module proxy)。direct 指示 go 命令应该与版本控制系统通信。off 表示不应尝试通信。GOPRIVATE 和 GONOPROXY 环境变量也可用于控制此行为。

对于 GOPROXY 列表中的每个条目，go 命令请求可能提供这个包的每个模块路径的最新版本（即包路径的每个前缀）。对于每一个请求成功的模块路径，go 命令都会下载最新版本的该模块，并检查该模块是否包含请求的包。如果一个或多个模块都包含请求的包，则使用路径最长的那个模块。如果找到一个或多个模块但其中没有一个包含请求的包，则会报告一个错误。如果没有找到模块，go 命令会尝试 GOPROXY 列表中的下一个条目。如果没有剩余条目了，则报告一个错误。

例如，假设 go 命令正在寻找提供包 golang.org/x/net/html 的模块，并且 GOPROXY 设置为 https://corp.example.com,https://proxy.golang.org。go 命令可能会发出以下请求：

- 到 https://corp.example.com/（并行）：

请求最新版本的 golang.org/x/net/html
请求最新版本的 golang.org/x/net
请求最新版本的 golang.org/x
请求最新版本的 golang.org

- 如果对 https://corp.example.com/的所有请求都以 404 或 410 失败，则到 https://proxy.golang.org/：

请求最新版本的 golang.org/x/net/html
请求最新版本的 golang.org/x/net
请求最新版本的 golang.org/x

请求最新版本的 golang.org

找到合适的模块后，go 命令会在主模块的 go.mod 文件中添加一个新的依赖项(requirement)以及新模块的路径和版本。这样可以确保以后加载同一个包时，会使用同一个版本的同一个模块。如果解析的包不是由主模块中的包导入的（其他模块导入的），则新依赖项将具有// indirect 注释。

go.mod 文件

模块由其根目录中名为 go.mod 的 UTF-8 编码的文本文件定义。go.mod 文件是基于行的(line-oriented)。每行包含一个指令，由关键字和参数组成。例如：

```
module example.com/my/thing

go 1.12

require example.com/other/thing v1.0.2
require example.com/new/thing/v2 v2.3.4
exclude example.com/old/thing v1.2.3
replace example.com/bad/thing v1.4.5 => example.com/good/thing v1.4.5
retract [v1.9.0, v1.9.5]
```

前导关键字可以从相邻行中分离出来以创建一个块，就像在 Go 导入(import)中一样。

```
require (
    example.com/new/thing/v2 v2.3.4
    example.com/old/thing v1.2.3
)
```

go.mod 文件被设计为人类可读和机器可写。go 命令提供了几个更改 go.mod 文件的子命令。例如，go get 可以升级或降级指定的依赖项。加载模块图(module graph)的命令会在需要时自动更新 go.mod。go mod edit 可以执行低级(low-level)编辑。Go 程序可以使用 golang.org/x/mod/modfile 包以编程方式进行相同的更改。

主模块(main module)和使用一个本地文件路径(a local file path)指定的任何替换模块(replacement module)都需要 go.mod 文件。但是，缺少显式 go.mod 文件的模块可能仍需要作为依赖项，或者用作指定的一个模块路径和版本的替换；请参阅与非模块存储库的兼容性。

词法元素(Lexical elements)

解析 go.mod 文件时，其内容被分解为一系列词法单元(token)。有几种标记：空白(White space)、注释(comments)、标点符号(punctuation)、关键字(keywords)、标

识符(identifiers)和字符串(strings)。

空白(White space)由空格 (U+0020)、制表符 (U+0009)、回车 (U+000D) 和换行符 (U+000A) 组成。除了换行符之外的空白字符没有任何特殊作用，只是用来分隔一个个词法单元。换行符是重要的词法单元。

注释(comments)以//开头，一直到行尾。/* */不允许被使用。

标点符号(punctuation)包括(,)和=>。

关键字区分 go.mod 文件中不同类型的指令。允许的关键字是 module、go、require、replace、exclude 和 retract。

标识符(identifiers)是非空白字符(non-whitespace character)的序列，例如模块路径或语义版本号。

字符串(strings)是引号包起来的字符序列。有两种字符串：以引号开头和结尾的解释型字符串（“，U+0022）和以重音符号（`，U+0060）开头和结尾的原始字符串。解释型字符串可能包含由反斜杠 (\, U+005C) 后跟另一个字符组成的转义序列。转义引号 (\") 不会终止解释型字符串。解释型字符串的未引用值（未使用引号包起来的值）是引号之间的字符序列，其中每个转义序列由反斜杠后面的字符替换（例如，\"被替换为", \n 被替换为 n)。相反，原始字符串的未引用值（未使用引号包起来的值）只是重音符号之间的字符序列；反斜杠在原始字符串中没有特殊含义。

标识符和字符串在 go.mod 语法中是可以互换的。

模块路径和版本号(Module paths and versions)

go.mod 文件中的大多数标识符和字符串都是模块路径或版本号。

模块路径必须满足以下要求：

- 路径必须由使用一个或多个由斜杠（/、U+002F）分隔的路径元素组成。它不能以斜杠开头或结尾。
- 每个路径元素都是由 ASCII 字母、ASCII 数字和有限的 ASCII 标点符号（-、.、_和~）组成的非空字符串。
- 路径元素不能以点(., U+002E)开始或结束。
- 直到第一个点号的路径元素前缀不能是 Windows 上的保留文件名，无论大小写如何（CON、com1、NuL 等）。
- 直到第一个点号的路径元素前缀不得以波浪号结尾，后跟一个或多个数字（如 EXAMPL~1.COM）。

如果模块路径出现在 require 指令中并且没有被替换，或者模块路径出现在 replace 指令的右侧，则 go 命令可能需要使用该路径来下载模块，并且必须满足一些附

加要求。

- 前导路径元素（直到第一个斜杠，如果有的话），按照惯例是一个域名，只能包含小写 ASCII 字母、ASCII 数字、点号（.，U+002E）和破折号（-，U+002D）；它必须至少包含一个点号，并且不能以破折号开头。
- 对于/vN 形式的最终路径元素，其中 N 看起来应该是数字（ASCII 数字和点号），N 不能以前导零开头，也不能是/v1，并且不能包含任何点号（译者注：这里指的应该是不能以任何个数的点号开头）。
  对于以 gopkg.in/开头的路径，此要求被替换为路径遵循 gopkg.in 服务约定的要求。

go.mod 文件中的版本号可能是规范的(canonical)或不规范的。

规范版本号以字母 v 开头，随后是遵循语义版本号控制 2.0.0(Semantic Versioning 2.0.0)规范的语义版本号。有关详细信息，请参阅版本号。

大多数其他标识符和字符串可以用作非规范版本号，尽管有一些限制可以避免文件系统、存储库和模块代理出现问题。非规范版本号只允许用在主模块的 go.mod 文件中。go 命令在自动更新 go.mod 文件时会尝试用等效的规范版本号替换每个非规范版本号。

在模块路径与版本号相关联的地方（如在 require、replace 和 exclude 指令中），最终的路径元素必须与版本号一致。请参阅主版本号后缀。

语法(Grammar)

go.mod 语法在下面使用扩展巴科斯-瑙尔范式 (EBNF) 指定。有关 EBNF 语法的详细信息，请参阅 Go 语言规范中的符号部分。

```
GoMod = { Directive } .
Directive = ModuleDirective |
            GoDirective |
            RequireDirective |
            ExcludeDirective |
            ReplaceDirective |
            RetractDirective .
```

换行符、标识符和字符串分别用 newline、ident 和 string 表示。

模块路径和版本号用 ModulePath 和 Version 表示。

```
ModulePath = ident | string . /* see restrictions above */
Version = ident | string .      /* see restrictions above */
```

模块指令(module directive)

模块指令定义主模块的路径。go.mod 文件必须只包含一个模块指令。
ModuleDirective = "module" (ModulePath | "(" newline ModulePath newline ")") newline .

例如：
module golang.org/x/net

弃用
模块可以在段落开头的注释块中包含字符串 Deprecated:（区分大小写）标记为已弃用。弃用消息在冒号之后开始并运行到段落的末尾。注释可以立即出现在模块指令之前或之后的同一行。

例如：
// Deprecated: use example.com/mod/v2 instead.
module example.com/mod

从 Go 1.17 开始，go list -m -u 检查构建列表中所有已弃用模块的信息。go get 检查在命令行给出的构建的包所需的过时模块。

当 go 命令检索模块的弃用信息时，它会从匹配@latest 版本号查询(version query)的版本加载 go.mod 文件，而不考虑撤回或排除。go 命令从同一个 go.mod 文件加载撤回。

要弃用模块，作者可以添加// Deprecated:注释并发布一个新版本标签。作者可能会在更高版本中更改或删除弃用消息。

弃用适用于模块的所有次版本号。为此，高于 v2 的主版本号被视为单独的模块，因为它们的主版本号后缀为它们提供了不同的模块路径。

弃用消息旨在通知用户该模块不再受支持，并提供迁移说明，例如，迁移到最新的主版本。不能弃用单个次版本和补丁版本；撤回(retract)可能更适合。

go 指令(go directive)

go 指令假设这个模块是由指定语义版本号的 Go 版本编写的。版本号必须是有效的 Go 发行版本：一个正整数后跟一个点和一个非负整数（例如，1.9、1.14）。

go 指令最初旨在支持对 Go 语言的向后不兼容的更改（请参阅 Go 2 transition）。自从引入模块以来，没有发生不兼容的语言更改，但 go 指令仍然影响新语言功能的使用：

- 对于模块中的包，编译器拒绝使用 go 指令指定的版本之后引入的语言特性。例如，如果一个模块有指令 go 1.12，它的包可能不会使用像 1_000_000 这样

的数字字面量，因为这是在 Go 1.13 中引入的特性。

- 如果使用较旧的 Go 版本构建模块的包之一并遇到编译错误，该错误表明该模块是使用较新的 Go 版本编写的。例如，假设一个模块有指令 go 1.13，而一个包使用了数字字面量 1_000_000。如果该包使用 Go 1.12 构建，那么编译器会注意到代码是为 Go 1.13 编写的。

此外，go 命令会根据 go 指令指定的版本更改其行为。这具有以下效果：

- 在 go 1.14 或更高版本中，可以启用自动（vendor 模式）vendoring。如果文件 vendor/modules.txt 存在并且与 go.mod 一致，则无需显式使用-mod=vendor 标志。
- 在 go 1.16 或更高版本中，all 包模式仅匹配由主模块中的包和测试(test)级联导入的包。这是自引入模块以来 go mod vendor 保留的同一组包。在较低版本中，all 还包含测试用包，这些包是主模块中导入的、测试用导入的等等。
- 在 go 1.17 或更高版本：
  - go.mod 文件为每个模块包含一个显式的 require 指令，该指令提供由主模块中的包或测试(test)级联导入的任何包。（在 go 1.16 及更低版本中，仅当最小版本选择(minimal version selection)会选择不同的版本时才包含间接依赖关系。）此额外信息启用模块图修剪(module graph pruning)和延迟模块加载(lazy module loading)。
  - 因为可能比以前的 go 版本有更多// indirect 依赖项，所以间接依赖项记录在 go.mod 文件中的单独块(block)中。
  - go mod vendor 省略了 go.mod 和 go.sum 文件以获取 vendored 依赖项。（这允许在 vendor 的子目录中调用 go 命令来识别正确的主模块。）
  - go mod vendor 将每个依赖项的 go.mod 文件中的 go 版本号记录在 vendor/modules.txt 中。

一个 go.mod 文件最多可以包含一个 go 指令。如果未指定 Go 版本，大多数命令将添加一个带有当前 Go 版本号的 go 指令。

从 Go 1.17 版本开始，如果缺少 go 指令，则假定为 go 1.16。

```
GoDirective = "go" GoVersion newline .
GoVersion = string | ident .    /* valid release version; see above */
```

例如：

```
go 1.14
```

依赖指令(require directive)

require 指令声明给定模块依赖项的最低版本需求。对于每个所需的模块版本，go 命令加载该版本的 go.mod 文件并合并该文件中的依赖项。加载完所有依赖项后，go 命令使用最小版本选择 (MVS) 来解析它们以生成构建列表(build list)。

go 命令会自动为某些依赖项添加// indirect 注释。// indirect 注释表示主模块（译者注：当前 go.mod 文件所在模块）中的任何包都没有直接导入所依赖模块中的

包。

如果 go 指令指定 go 1.16 或更低版本，则当所选模块的版本高于主模块的其他依赖项已经暗示的（传递性的）版本时，go 命令会添加间接依赖注释（译者注：即 // indirect）。这可能是因为显式升级（go get -u ./...）、删除了之前强加的其他一些依赖项（go mod tidy），或者导入了一个在它自己的 go.mod 文件里没有相应的依赖项的包（例如一个没有自己的 go.mod 文件的依赖项）。

在 go 1.17 及更高版本中，go 命令为每个模块添加了一个间接依赖注释，这些模块提供由主模块中的包或测试(test)导入（甚至间接地）的任何包，或作为参数传递给 go get。这些更全面的要求使模块图修剪(module graph pruning)和模块懒加载(lazy module loading)成为可能。

```
RequireDirective = "require" ( RequireSpec | "(" newline { RequireSpec } ")" newline ) .
RequireSpec = ModulePath Version newline .
```

示例：

```
require golang.org/x/net v1.2.3

require (
	golang.org/x/crypto v1.4.5 // indirect
	golang.org/x/text v1.6.7
)
```

排除指令(exclude directive)

exclude 指令防止 go 命令加载某个模块版本。

从 Go 1.16 开始，如果任何 go.mod 文件中的 require 指令引用的版本，被主模块的 go.mod 文件中的 exclude 指令排除，则忽略该依赖项。这可能会导致 go get 和 go mod tidy 等命令在 go.mod 文件里添加更高版本的依赖项，并在适当的情况下使用// indirect 注释。

在 Go 1.16 之前，如果 require 指令引用了排除的版本，go 命令会列出模块的可用版本（正如 go list -m -versions 所示）并加载下一个更高的非排除版本。这可能会导致不确定的版本选择，因为下一个更高版本可能会随着时间而改变。语义版本号中的发布(release)版本号和预发布(pre-release)版本号就是为此考虑，但不包括伪版本号。如果没有更高版本，go 命令就报错。

exclude 指令仅适用于主模块的 go.mod 文件，在其他模块中被忽略。有关详细信息，请参阅最小版本号选择(Minimal version selection)。

```
ExcludeDirective = "exclude" ( ExcludeSpec | "(" newline { ExcludeSpec } ")" newline ) .
ExcludeSpec = ModulePath Version newline .
```

例如：

```
exclude golang.org/x/net v1.2.3

exclude (
	golang.org/x/crypto v1.4.5
	golang.org/x/text v1.6.7
)
```

替换指令(replace directive)

替换指令(replace)使用在别处找到的内容替换模块的特定版本或模块的所有版本的内容。可以使用另一个模块路径和版本号或特定于平台的文件路径来指定替换(replacement)。

如果箭头左侧的模块存在版本号(=>)，则仅替换该模块的特定版本，其他版本还是正常访问。如果省略左侧版本号，则替换该模块的所有版本。

如果箭头右侧的路径是绝对或相对路径（以./或../开头），则解释为替换模块的根目录的本地文件路径，它必须包含 go.mod 文件。在这种情况下必须省略替换模块的版本号。

如果右侧的路径不是本地路径，则必须是有效的模块路径。在这种情况下，需要指定替换模块的版本号。并且相同的模块版本不得同时出现在构建列表中。

不管替换是用本地路径还是模块路径指定，如果替换模块有一个 go.mod 文件，它的模块指令(module directive)必须匹配它用来替换的模块路径。

replace 指令仅适用于主模块的 go.mod 文件，在其他模块中被忽略。有关详细信息，请参阅最小版本号选择(Minimal version selection)。

如果有多个主模块，则替换应用于所有主模块的 go.mod 文件。不允许跨主模块冲突的 replace 指令，并且必须在 go.work 文件的替换(replace in the go.work file)中删除或覆盖。

请注意，单独的 replace 指令不会将模块添加到模块图(module graph)中。在主模块的 go.mod 文件或依赖项的 go.mod 文件中，都可以有指向被替换的模块版本的 require 指令。如果没有使用到左侧的模块版本，则 replace 指令不起作用。

```
ReplaceDirective = "replace" ( ReplaceSpec | "(" newline { ReplaceSpec } ")" newline ) .
ReplaceSpec = ModulePath [ Version ] "=>" FilePath newline
            | ModulePath [ Version ] "=>" ModulePath Version newline .
FilePath = /* platform-specific relative or absolute file path */
```

例如：

```
replace golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5

replace (
    golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5
    golang.org/x/net => example.com/fork/net v1.4.5
    golang.org/x/net v1.2.3 => ./fork/net
    golang.org/x/net => ./fork/net
)
```

撤回指令(retract directive)

retract 指令表示不应该依赖 go.mod 定义的模块的版本号或版本号范围。当版本过早发布或版本发布后发现严重问题时，retract 指令很有用。撤回的版本应该在版本控制存储库和模块代理(module proxies)中保持可用，以确保依赖它们的构建不会被破坏。撤回(retract)这个词是从学术文献中借来的：一篇被撤回的研究论文仍然可用，但它有问题，不应该成为未来工作的基础。

当模块版本被撤回时，用户使用 go get、go mod tidy 或其他命令时不会自动升级它。依赖于撤回版本的构建应该能继续工作，但是当用户使用 go list -m -u 检查更新或使用 go get 更新相关模块时，用户将收到撤回通知。

要撤回一个版本，模块作者应该向 go.mod 添加一个撤回指令，然后发布一个包含该指令的新版本。 新版本必须高于其他发布或预发布版本；也就是说，@latest 版本查询(version query)应该在考虑撤回之前解析为新版本。go 命令从 go list -m -retracted $modpath@latest 显示的版本加载并应用撤回（其中$modpath 是模块路径）。

除非使用-retracted 标志，否则撤回的版本从 go list -m -versions 打印的版本列表中隐藏。解析@>=v1.2.3 或@latest 之类的版本查询时，也将排除撤回的版本。

包含撤回指令的版本可以撤回自己。如果模块的最高版本或预发布版本撤回自己，则在排除撤回的版本后，@latest 查询将解析为较低版本。

例如，考虑模块 example.com/m 的作者意外发布版本 v1.0.0 的情况。为了防止用户升级到 v1.0.0，作者可以在 go.mod 中添加两个 retract 指令，然后用撤回信息标记 v1.0.1。

```
retract (
    v1.0.0 // Published accidentally.
    v1.0.1 // Contains retractions only.
)
```

当用户运行 go get example.com/m@latest 时，go 命令从 v1.0.1 读取撤回信息，它现在是最高版本。v1.0.0 和 v1.0.1 都被撤回，因此 go 命令将升级（或降级！）到下一个最高版本，可能是 v0.9.5。

retract 指令可以使用单个版本号（如 v1.0.0）或具有上限和下限的版本号闭区间来编写，由[和]分隔（如[v1.1.0，v1.2.0]）。单个版本号相当于上下限相同的区间。与其他指令一样，多个 retract 指令可以组合在一个由在行尾的(和)分隔的块中。

每个 retract 指令都应该有一个注释来解释撤回的理由，尽管这不是强制性的。go 命令可能会在撤回版本的相关警告和 go list 的输出中显示撤回理由注释。撤回理由注释可以直接写在撤回指令的上方（中间没有空行），也可以写在同一行之后。如果撤回理由注释出现在块上方，它适用于块内没有自己注释的所有撤回指令。撤回理由注释可以跨越多行。

```
RetractDirective = "retract" ( RetractSpec | "(" newline { RetractSpec } ")" newline ) .
RetractSpec = ( Version | "[" Version "," Version "]" ) newline .
```

例如：

```
retract v1.0.0
retract [v1.0.0, v1.9.9]
retract (
	v1.0.0
	[v1.0.0, v1.9.9]
)
```

retract 指令是在在 Go 1.16 中添加。如果在主模块(main module)的 go.mod 文件中写入 retract 指令，Go 1.15 及更低版本将报告错误，并且将忽略 go.mod 文件中的依赖项的 retract 指令。

原子更新(Automatic updates)

如果 go.mod 缺少信息或不能准确反映现实，大多数命令都会报告错误。go get 和 go mod tidy 命令可用于解决大多数此类问题。此外，-mod=mod 标志可以与大多数模块敏感命令（go build、go test 等）一起使用，以指示 go 命令自动修复 go.mod 和 go.sum 中的问题。

例如，考虑这个 go.mod 文件：

```
module example.com/M

go 1.16

require (
	example.com/A v1
	example.com/B v1.0.0
	example.com/C v1.0.0
	example.com/D v1.2.3
	example.com/E dev
)
```

exclude example.com/D v1.2.3

使用-mod=mod 触发的更新将非规范版本标识符重写为规范形式，因此 example.com/A 的 v1 变为 v1.0.0，example.com/E 的 dev 成为 dev 分支上最新提交的伪版本，也许是 v0.0.0-20180523231146-b3f5c0f6e5f1。（译者注：像 example.com/F/v6 latest 的 latest 会被重写为主版本号为 6 的最新稳定版本号，也许是 v6.2.1）

此更新修改了依赖项以符合排除项的要求，因此对排除的 example.com/D v1.2.3 这个依赖项将被更新为使用 example.com/D 的下一个可用版本，可能是 v1.2.4 或 v1.3.0。

此更新删除了多余或误导性的依赖项。例如，如果 example.com/A v1.0.0 本身需要 example.com/B v1.2.0 和 example.com/C v1.0.0，那么 go.mod 对 example.com/B v1.0.0 的需求具有误导性（将被 example.com/A 需要的 v1.2.0 取代），并且其对 example.com/C v1.0.0 的需求是多余的（被 example.com/A 需要相同版本所暗示），因此两者都将被删除。如果主模块包含直接从 example.com/B 或 example.com/C 模块导入包的包，则将保留对它们的依赖项，但会更新为正在使用的实际版本。

最后，更新以规范格式重新格式化 go.mod 文件的内容，以便未来的机械更改将导致最小的差异。如果只需要更改格式，go 命令不会更新 go.mod 文件的内容。

因为模块图定义了 import 语句的含义，所以任何加载包的命令也使用 go.mod 文件，因此可以更新它，包括 go build、go get、go install、go list、go test、go mod tidy。

在 Go 1.15 及更低版本中，默认启用-mod=mod 标志，因此会自动执行更新。从 Go 1.16 开始，go 命令的作用就像设置了-mod=readonly 一样：如果需要对 go.mod 进行任何更改，go 命令会报告错误并建议修复。

最小版本选择(Minimal version selection, MVS)

Go 使用一种称为最小版本选择(Minimal version selection, MVS) 的算法来选择一组模块版本号以在构建包时使用。Russ Cox 在 Minimal Version Selection 中详细描述了 MVS 算法。

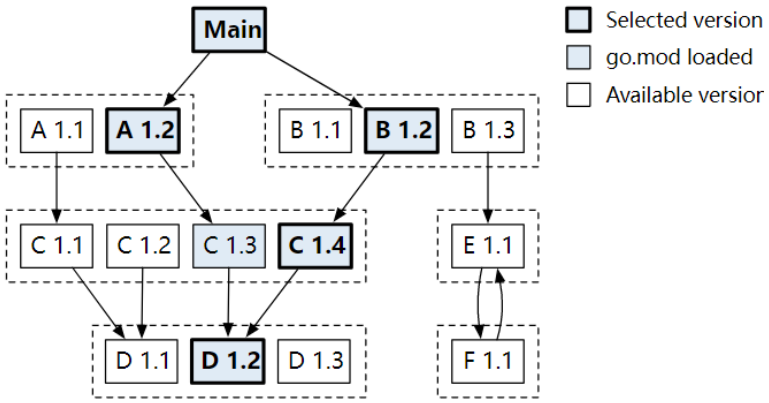
从概念上讲，MVS 在模块的有向图上运行，由 go.mod 文件指定。图中的每个顶点代表一个模块版本。每条边表示依赖项的最低要求版本，使用 require 指令指定。该图可以通过主模块的 go.mod 文件中的 exclude 和 replace 指令以及 go.work 文件中的 replace 指令来修改。

MVS 生成构建列表(build list)作为输出，模块版本列表用于构建。

MVS 从主模块（图中没有版本的特殊顶点）开始并遍历图，跟踪每个模块所需的最高版本。在遍历结束时，最高需求的版本构成构建列表：它们是满足所有需求的最低版本。

可以使用命令 go list -m all 检查构建列表。与其他依赖管理系统不同，构建列表不保存在“lock”文件中。MVS 是确定性的，并且当依赖项发布新版本时构建列表也不会改变（译者注：毕竟我们在 go.mod 文件里明确指定了依赖项的版本号），因此 MVS 在每个与模块相关命令的开头计算它。

考虑下图中的示例。主模块需求 1.2 或更高版本的模块 A 和 1.2 或更高版本的模块 B。A 1.2 和 B 1.2 分别需求 C 1.3 和 C 1.4。C 1.3 和 C 1.4 都需求 D 1.2。



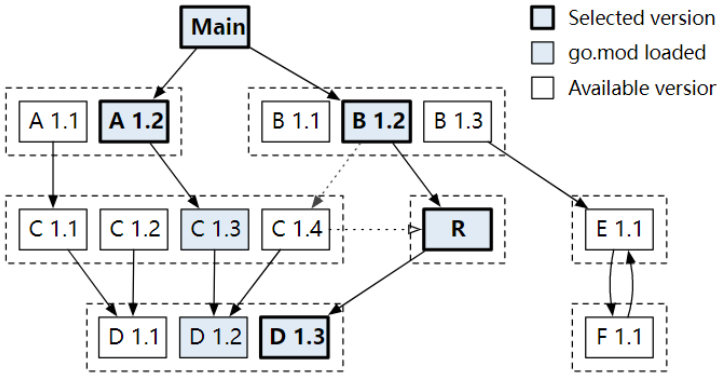
MVS 访问并加载 go.mod 文件，得到每个以蓝色突出显示的模块版本节点。在图遍历结束时，MVS 返回包含粗体版本的构建列表：A 1.2、B 1.2、C 1.4 和 D 1.2。请注意，可以使用更高版本的 B 和 D，但 MVS 不会选择它们，因为没有任何东西依赖它们。

替换(Replacement)

可以使用主模块的 go.mod 文件或工作区的 go.work 文件中的 replace 指令替换模块的内容（包括其 go.mod 文件）。replace 指令可以应用于模块的特定版本或模块的所有版本。

替换会更改模块图，因为替换模块可能具有与被替换版本不同的依赖关系。

考虑下面的示例，其中 C 1.4 已替换为 R。R 依赖于 D 1.3 而不是 D 1.2，因此 MVS 返回包含 A 1.2、B 1.2、C 1.4（替换为 R）和 D 1.3 的构建列表。

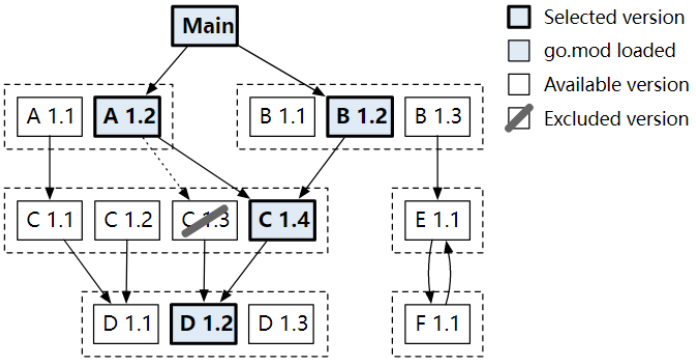


排除(Exclusion)

也可以在主模块的 go.mod 文件中使用 exclude 指令排除某个模块的特定版本。

排除也会更改模块图。当一个版本被排除时，它会从模块图中被移除，并且对它的需求被重定向到它的下一个更高版本。

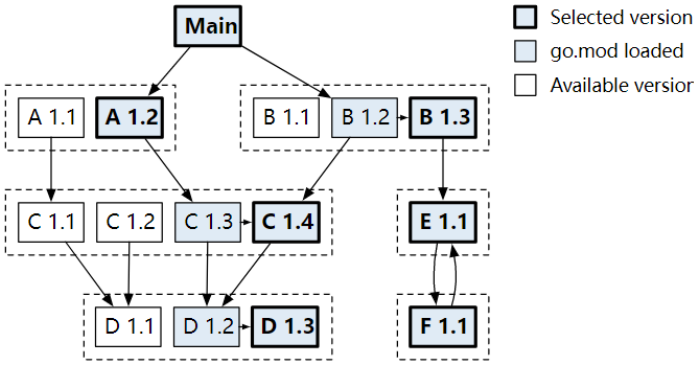
考虑下面的例子。C 1.3 已被排除在外。MVS 将表现得好像 A 1.2 需求 C 1.4（下一个更高版本）而不是 C 1.3。



升级(Upgrades)

go get 命令可用于升级一组模块。要执行升级，go 命令会在运行 MVS 之前更改模块图，方法是将已访问版本的边添加到升级版本。

考虑下面的例子。模块 B 可以从 1.2 升级到 1.3，C 可以从 1.3 升级到 1.4，D 可以从 1.2 升级到 1.3。



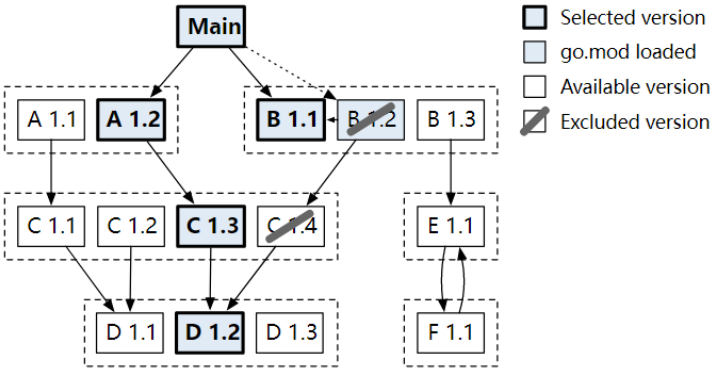
升级（和降级）可能会添加或删除间接依赖项。在这种情况下，E 1.1 和 F 1.1 出现在升级后的构建列表中，因为 B 1.3 需求 E 1.1。

为了保留升级，go 命令更新 go.mod 中的依赖。 它将对 B 的需求更改为 1.3 版。它还将使用// indirect 注释添加对 C 1.4 和 D 1.3 的需求，因为不这么做的话就不会选择这些版本。

降级(Downgrade)

go get 命令也可用于降级一组模块。要执行降级，go 命令通过删除降级版本之上的版本来更改模块图。它还会删除依赖于已删除版本的其他模块版本，因为它们可能与其依赖的模块的降级版本不兼容。如果主模块需求一个因为降级而删除的模块版本，则将该需求更改为尚未删除的先前版本。 如果没有先前的版本可用，则放弃该需求。

考虑下面的例子。假设发现 C 1.4 有问题，所以我们降级到 C 1.3。C 1.4 从模块图中移除。B 1.2 也被删除，因为它需求 C 1.4 或更高版本。主模块对 B 的需求改为 1.1 版本。



go get 还可以完全删除依赖项，在参数后使用@none 后缀。这与降级类似。@none 命名的模块的所有版本都将从模块图中删除。

模块依赖图化简(Module graph pruning)

如果主模块是 go 1.17 或更高版本，则用于最小版本选择的模块图(module graph)立即需求(immediate requirements)的依赖项模块仅包括那些在它们自己的 go.mod 文件中指定 go 1.17 或更高版本的模块，除非该模块的版本也是（传递地）被其他 go 1.16 或更低版本的依赖项所需求。（go 1.17 依赖项的传递依赖项被从模块图中删除。）

由于 go 1.17 的 go.mod 文件包含该模块中构建任何包或测试所需的每个依赖项的 require 指令，因此化简后的模块图包含 go build 或 go test 在主模块里显式需求的任何依赖项对应的包所需要的所有依赖项。一个模块不被构建或测试一个给定模块里的任何包所需要，就不会影响这些包在运行时的行为，因此从模块图中删除的依赖关系只会在其他不相关的模块之间造成干扰。

依赖关系被删除的模块仍然可以出现在模块图中，并且仍然由 go list -m all 报告：它们选择的版本(selected versions)是已知且明确定义的，并且可以从这些模块中加载包（例如，作为测试的传递依赖项从其他模块加载的模块）。但是，由于 go 命令无法轻松识别这些模块的哪些依赖项是合适的，因此 go build 和 go test 的参数不能包含已被删除需求的模块的包。go get 将包含每个命名包的模块提升为显式的依赖项，从而允许在该包上调用 go build 或 go test。

由于 Go 1.16 及更早版本不支持模块图化简，因此对于指定 go 1.16 或更低版本的每个模块，仍然包含依赖项的完全传递闭包（包括传递 go 1.17 依赖项）。（在 go 1.16 及以下版本中，go.mod 文件仅包含直接依赖项(direct dependencies)，因此必须加载更大的模块图以确保包含所有间接依赖项。）

go mod tidy 为模块记录的 go.sum 文件默认包含低于其 go 指令中指定的 Go 版本所需的校验值。因此，go 1.17 模块包含 Go 1.16 加载的完整模块图所需的校验值，但 go 1.18 模块将仅包含 Go 1.17 加载的化简后的模块图所需的校验值。-compat 标志可用于覆盖默认版本（例如，在 go 1.17 模块中更激进地化简 go.sum 文件）。

有关详细信息，请参阅设计文档。

模块懒加载(Lazy module loading)

为模块图化简添加的更全面的需求还支持在某个模块内工作时进行另一项优化。如果主模块位于 go 1.17 或更高版本，则 go 命令会避免加载完整的模块图，直到（除非）需要它。相反，它只加载主模块的 go.mod 文件，然后尝试加载仅使用这些需求构建的包。如果在这些需求中找不到要导入的包（例如，用于主模块外部包测试的依赖项），则按需加载模块图的其余部分。

如果在不加载模块图的情况下可以找到所有导入的包，则 go 命令只为包含这些包的模块加载 go.mod 文件，并根据主模块的需求检查它们的需求以确保它们在

本地一致。（由于版本控制合并、手动编辑以及已使用本地文件系统路径替换的模块中的更改，可能会出现不一致。）

工作空间(Workspaces)

工作区(workspace)是磁盘上的模块集合，在运行最小版本选择(MVS)时用作根模块。

工作空间可以在 go.work 文件中声明，该文件指定工作空间中每个模块的模块目录的相对路径。当不存在 go.work 文件时，工作区由包含当前目录的单个模块组成。

大多数与模块一起使用的 go 子命令在由当前工作空间确定的模块集上运行。go mod init、go mod why、go mod edit、go mod tidy、go mod vendor 和 go get 始终在单个主模块上运行。

命令通过首先检查-workfile 标志来确定它是否在工作区的上下文中。如果-workfile 设置为 off，则该命令将位于单模块的上下文中。如果它为空或未提供，该命令将搜索当前工作目录，然后是后续父目录，以查找文件 go.work。如果找到一个文件，该命令将在它定义的工作空间中运行；否则，工作区将仅包括包含工作目录的模块。如果-workfile 命名一个以.work 结尾的现有文件的路径，则将启用工作区模式。任何其他值都是一个错误。

go.work 文件(go.work files)

工作空间由名为 go.work 的 UTF-8 编码的文本文件定义。go.work 文件是面向行的。每行包含一个指令，由关键字和参数组成。例如：

```
go 1.18

use ./my/first/thing
use ./my/second/thing

replace example.com/bad/thing v1.4.5 => example.com/good/thing v1.4.5
```

与 go.mod 文件一样，可以从相邻行中分解出前导关键字来创建块。

```
use (
    ./my/first/thing
    ./my/second/thing
)
```

go 命令提供了几个用于操作 go.work 文件的子命令。go work init 创建新的 go.work 文件。go work use 将模块目录添加到 go.work 文件中。go work edit 执行低级编辑。Go 程序可以使用 golang.org/x/mod/modfile 包以编程方式进行相同的更改。

词法元素(Lexical elements)

go.work 文件中的词法元素的定义方式与 go.mod 文件的完全相同。

文法(Grammar)

go.work 语法在下面使用扩展巴科斯-瑙尔形式(EBNF)指定。有关 EBNF 语法的详细信息，请参阅 Go 语言规范中的符号部分。

```
GoWork = { Directive } .
Directive = GoDirective |
                    UseDirective |
                    ReplaceDirective .
```

换行符、标识符和字符串分别用 newline、ident 和 string 表示。

模块路径和版本用 ModulePath 和 Version 表示。模块路径和版本的指定方式与 go.mod 文件的完全相同。

```
ModulePath = ident | string . /* see restrictions above */
Version = ident | string .       /* see restrictions above */
```

go 指令(go directive)

有效的 go.work 文件中需要 go 指令。版本必须是有效的 Go 发行版本：一个正整数后跟一个点和一个非负整数（例如，1.18、1.19）。

go 指令指示 go.work 文件打算使用的 go 工具链的版本。如果对 go.work 文件格式进行了更改，工具链的未来版本将根据其指示的版本来解释该文件。

一个 go.work 文件最多可以包含一个 go 指令。

```
GoDirective = "go" GoVersion newline .
GoVersion = string | ident .    /* valid release version; see above */
```

例如：

```
go 1.18
```

use 指令(use directive)

use 将磁盘上的模块添加到工作空间中的主模块集合里。它的参数是相对于包含模块的 go.mod 文件的目录的路径。use 指令不会添加包含在其参数目录的子目录中的模块。这些模块可以通过单独的 use 指令包含其 go.mod 文件所在目录的路径来添加。

UseDirective = "use" (UseSpec | "(" newline { UseSpec } ")" newline) .
UseSpec = FilePath newline .
FilePath = /* platform-specific relative or absolute file path */

例如：

```
use ./mymod    // example.com/mymod

use (
        ../othermod
        ./subdir/thirdmod
)
```

替换指令(replace directive)

与 go.mod 文件中的 replace 指令类似，go.work 文件中的 replace 指令将模块的特定版本或模块的所有版本的内容替换为其他地方的内容。go.work 中的通配符替换会覆盖(override)go.mod 文件中特定于版本的 replace。

go.work 文件中的 replace 指令会覆盖工作空间模块中被替换的任何相同模块或模块版本。

ReplaceDirective = "replace" (ReplaceSpec | "(" newline { ReplaceSpec } ")" newline) .
ReplaceSpec = ModulePath [Version] "=>" FilePath newline
| ModulePath [Version] "=>" ModulePath Version newline .
FilePath = /* platform-specific relative or absolute file path */

例如：

```
replace golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5

replace (
        golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5
        golang.org/x/net => example.com/fork/net v1.4.5
        golang.org/x/net v1.2.3 => ./fork/net
        golang.org/x/net => ./fork/net
)
```

与非模块存储库的兼容性(Compatibility with non-module repositories)

为了确保从 GOPATH 到模块的平滑过渡，go 命令可以通过添加 go.mod 文件从尚未迁移到模块的存储库中以模块感知模式(module-aware mode)下载和构建包。

当 go 命令直接从存储库下载给定版本的模块时，它会从存储库 URL 查找模块

路径，将版本号映射到存储库中的修订号(revision)，然后提取该修订号的存储库存档。如果模块的路径等于存储库根路径，并且存储库根目录不包含 go.mod 文件，则 go 命令会在模块缓存(module cache)中合成一个 go.mod 文件，该文件除了包含一个模块指令(module directive)，没有其他内容。由于合成的 go.mod 文件不包含其依赖项的 require 指令，因此依赖于它们的其他模块可能需要额外的 require 指令（带有// indirect 注释）以确保在每个构建中以相同版本号获取每个依赖项。

当 go 命令从代理(proxy)下载模块时，它将 go.mod 文件与模块内容的其余部分分开下载。如果原始模块没有，则代理预计会提供合成的 go.mod 文件。

+incompatible 版本(+incompatible versions)

主版本号 2 或更高版本号发布的模块必须在其模块路径上具有匹配的主版本号后缀。例如，如果一个模块在 v2.0.0 发布，它的路径必须有/v2 后缀。这允许 go 命令将项目的多个主版本视为不同的模块，即使它们是在同一个存储库中开发的。

在 go 命令中添加模块支持时引入了主版本号后缀要求，并且许多存储库在此之前已经标记（加标签）了主版本号 2 或更高版本的版本号。为了保持与这些存储库的兼容性，go 命令会在没有 go.mod 文件的主版本号 2 或更高版本的版本号中添加一个+incompatible 后缀。+incompatible 表示一个版本与具有较低主版本号的版本属于同一模块；因此，go 命令可能会自动升级到更高的+incompatible 版本，即使它可能会破坏构建。

考虑下面示例的需求：
require example.com/m v4.1.2+incompatible

版本 v4.1.2+incompatible 是指提供模块 example.com/m 的存储库中的语义版本标签 v4.1.2。该模块必须在存储库根目录中（即存储库根路径也必须是 example.com/m），并且不能有 go.mod 文件。该模块可能是具有较低主版本号的版本，例如 v1.5.2，并且 go 命令可能会从这些版本自动升级到 v4.1.2+incompatible（有关升级如何工作的信息，请参阅最小版本选择(MVS)）。

在版本 v2.0.0 被标记后迁移到模块的存储库通常应该发布一个新的主版本。在上面的示例中，作者应该使用路径 example.com/m/v5 创建一个模块，并且应该发布版本 v5.0.0。作者还应该更新模块中包的导入以使用前缀 example.com/m/v5 而不是 example.com/m。有关更详细的示例，请参阅 Go Modules: v2 and Beyond。

请注意，+incompatible 后缀不应出现在存储库中的标签上；v4.1.2+incompatible 之类的标签将被忽略。该后缀仅出现在 go 命令使用的版本号中。有关版本号和标签之间区别的详细信息，请参阅 Mapping versions to commits。

另请注意，+incompatible 后缀可能出现在伪版本号上。例如，v2.0.1-20200722182040-012345abcdef+incompatible 可能是有效的伪版本号。

最小的模块兼容性(Minimal module compatibility)

主版本号 2 或更高版本发布的模块需要在其模块路径上具有主版本号后缀。该模块可能会或可能不会在其存储库中的主版本号子目录中开发。当包导入使用 GOPATH 模式在模块里构建的包时，会有影响。

通常在 GOPATH 模式下，包存储在与其存储库的根路径加上里面的目录相匹配的路径中。例如，在子目录 sub 中具有根路径 example.com/repo 的存储库中的包将存储在$GOPATH/src/example.com/repo/sub 中，并将作为 example.com/repo/sub 导入。

对于具有主版本号后缀的模块，人们可能希望在 $GOPATH/src/example.com/repo/v2/sub 目录中找到包 example.com/repo/v2/sub。这将需要在其存储库的 v2 子目录中开发模块。go 命令支持这一点，但不需要它（请参阅 Mapping versions to commits）。

如果一个模块不是在主版本号子目录下开发的，那么它在 GOPATH 中的目录就不会包含主版本号后缀，并且它的包可能会在没有主版本号后缀的情况下被导入。在上面的示例中，该包将在目录$GOPATH/src/example.com/repo/sub 中找到，并将作为 example.com/repo/sub 导入。

这给打算同时在模块模式和 GOPATH 模式下构建包造成了问题：模块模式需要后缀，而 GOPATH 模式不需要。

为了解决这个问题，Go 1.11 中添加了最小的模块兼容性(minimal module compatibility)，并向后移植到 Go 1.9.7 和 1.10.3。当导入路径解析为 GOPATH 模式下的目录时：

- 解析$modpath/$vn/$dir 形式的导入时，其中：
  - $modpath 是一个有效的模块路径，
  - $vn 是主版本号后缀，
  - $dir 是一个可能为空的子目录，
- 如果以下所有条件都为真：
  - 软件包$modpath/$vn/$dir 不在任何相关的供应商目录(vendor directory)中。
  - go.mod 文件存在于与导入文件相同的目录中，或位于$GOPATH/src 根目录之前的任何父目录中，
  - 不存在 $GOPATH[i]/src/$modpath/$vn/$suffix 目录（对于任何根 $GOPATH[i]），
  - 文件$GOPATH[d]/src/$modpath/go.mod 存在（对于某些根$GOPATH[d]）并将模块路径声明为$modpath/$vn，
- 然后$modpath/$vn/$dir 的导入被解析到目录$GOPATH[d]/src/$modpath/$dir。

此规则允许已迁移到模块模式的包在以 GOPATH 模式构建时导入已迁移到模块模式的其他包，即使未使用主版本号子目录也可以。

模块感知命令(Module-aware commands)

大多数 go 命令可以在模块模式或 GOPATH 模式下运行。在模块模式下，go 命令使用 go.mod 文件来查找版本化的依赖项，它通常从模块缓存中加载包，如果缺少模块则下载模块。在 GOPATH 模式下，go 命令忽略模块；它在 vendor 目录和 GOPATH 中查找依赖项。

从 Go 1.16 开始，默认启用模块模式，无论是否存在 go.mod 文件。在较低版本中，当当前目录或任何父目录中存在 go.mod 文件时，会启用模块模式。

模块模式可以通过 GO111MODULE 环境变量进行控制，该变量可以设置为 on、off 或 auto。

- 如果 GO111MODULE=off，go 命令会忽略 go.mod 文件并以 GOPATH 模式运行。
- 如果 GO111MODULE=on 或未设置，则 go 命令以模块模式运行，即使不存在 go.mod 文件也是如此。并非所有命令都可以在没有 go.mod 文件的情况下工作：请参阅 Module commands outside a module。
- 如果 GO111MODULE=auto，如果当前目录或任何父目录中存在 go.mod 文件，则 go 命令以模块模式运行。在 Go 1.15 及更低版本中，这是默认行为。即使不存在 go.mod 文件，go mod 子命令和 go install 也会在模块模式下运行版本查询(version query)。

在模块模式下，GOPATH 不再定义为构建期间导入的含义，但它仍然存储下载的依赖项（在 GOPATH/pkg/mod 目录里；参见模块缓存(Module cache)）和安装的命令（在 GOPATH/bin 目录里，除非设置了 GOBIN 环境变量）。

构建命令(Build commands)

所有加载包信息的命令都是模块感知的。这包括：

- go build
- go fix
- go generate
- go get
- go install
- go list
- go run
- go test
- go vet

当在模块感知模式下运行时，这些命令使用 go.mod 文件来解释命令行上列出的或写在 Go 源文件中的导入路径。这些命令接受所有模块命令共有的以下标志。

- -mod 标志控制是否可以自动更新 go.mod 以及是否使用 vendor 目录。
  - -mod=mod 告诉 go 命令忽略 vendor 目录并自动更新 go.mod，例如，当一个导入的包不是任何已知模块提供的时。
  - -mod=readonly 告诉 go 命令忽略 vendor 目录并在 go.mod 需要更新时报告错误。
  - -mod=vendor 告诉 go 命令使用 vendor 目录。在这种模式下，go 命令不会使用网络或模块缓存。
  - 默认情况下，如果 go.mod 中的 go 版本是 1.14 或更高版本并且存在 vendor 目录，则 go 命令的行为就像使用了-mod=vendor 一样。否则，go 命令就像使用了-mod=readonly 一样。
- -modcacherw 标志指示 go 命令在模块缓存中创建具有读写权限的新目录，而不是将它们设为只读。如果一直使用此标志（通常通过在环境中设置环境变量 GOFLAGS=-modcacherw 或通过运行 go env -w GOFLAGS=-modcacherw），可以使用 rm -r 等命令删除模块缓存，而无需先更改权限。go clean -modcache 命令可用于删除模块缓存，无论是否使用了-modcacherw。
- -modfile=file.mod 标志指示 go 命令读取（并可能写入）一个替代模块根目录中的 go.mod 的文件。文件名必须以.mod 结尾。一个名为 go.mod 的文件必须仍然存在才能确定模块根目录，但它不会被访问。当指定-modfile 时，也会使用一个备用的 go.sum 文件：它的路径是从-modfile 标志派生的，通过修剪.mod 扩展名并附加.sum 扩展名得到。

vendor 模式(Vendoring)

使用模块时，go 命令通常通过将模块从其源下载到模块缓存中来满足依赖关系，然后从这些下载的模块拷贝中加载包。Vendor 模式可用于允许与旧版本的 Go 互操作，或确保用于构建的所有文件都存储在单个文件树中。

go mod vendor 命令在主模块的根目录中构建一个名为 vendor 的目录，其中包含在主模块中构建和测试包所需的所有包的拷贝。仅通过对主模块外部的包进行测试导入的包不包括在内。与 go mod tidy 和其他模块命令一样，构建 vendor 目录时不考虑除 ignore 之外的构建约束(build constraints)。

go mod vendor 还会创建文件 vendor/modules.txt，其中包含第三方提供的包的列表以及它们的模块版本。启用 vendor 模式时，此清单用作由 go list -m 和 go version -m 报告模块版本信息的来源。go 命令读取 vendor/modules.txt 时，会检查模块版本是否与 go.mod 一致。 如果在生成 vendor/modules.txt 后 go.mod 发生了变化，go 命令会报错。应该再次运行 go mod vendor 以更新 vendor 目录。

如果 vendor 目录存在于主模块的根目录中，如果主模块的 go.mod 文件中的 go 版本为 1.14 或更高版本，将自动使用 vendor 模式。要显式启用 vendor 模式，请使用标志-mod=vendor 调用 go 命令。要禁用 vendor 模式，请使用标志-mod=readonly 或-mod=mod。

启用 vendor 模式后，构建命令如 go build 和 go test 从 vendor 目录加载包，而不是访问网络或本地模块缓存。go list -m 命令仅打印有关 go.mod 中列出的模块的信息。go mod download 和 go mod tidy 等 go mod 命令在启用 vendor 模式时不会有不同的工作方式，并且仍会下载模块并访问模块缓存。go get 也不会在启用 vendor 模式时有所不同。

与 GOPATH 模式下的 vendor 模式不同，go 命令忽略主模块根目录以外位置的 vendor 目录。此外，由于不使用其他模块中的 vendor 目录，因此 go 命令在构建模块 zip 文件时不包括 vendor 目录（但请参阅已知错误#31562 和#37397）。

go get

用法：
go get [-d] [-t] [-u] [build flags] [packages]

例如：
# 升级一个指定的模块
$ go get -d golang.org/x/net

# 升级被主模块里的包导入包的模块
$ go get -d -u ./...

# 一个模块升级或降级到一个指定版本
$ go get -d golang.org/x/text@v0.3.2

# 一个模块升级到其 master 分支的最新 commit
$ go get -d golang.org/x/text@master

# 删除对一个模块的依赖并将依赖它的模块降级到不依赖它的版本
$ go get -d golang.org/x/text@none

go get 命令更新主模块的 go.mod 文件中的模块依赖项，然后构建和安装命令行中列出的包。

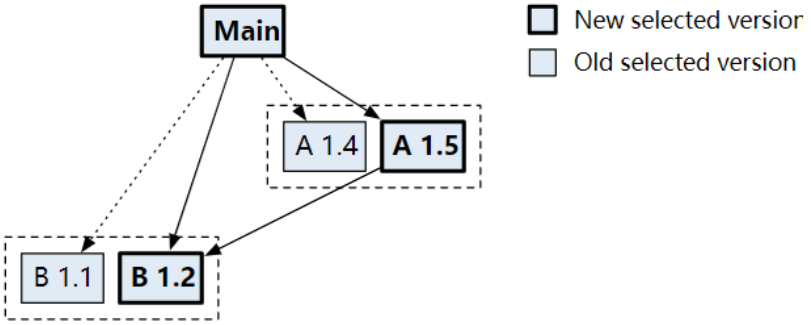
第一步是确定要更新哪些模块。go get 接受一个包、包模式(pattern)和模块路径的列表作为参数。如果指定了包参数，则 go get 更新提供该包的模块。如果指定了包模式（例如，all 或带有...通配符的路径），go get 将模式扩展为一组包，然后更新提供这些包的模块。如果一个参数命名一个模块而不是一个包（例如，模块 golang.org/x/net 在其根目录中没有包），go get 将更新模块但不会构建一个包。如果未指定任何参数，则 go get 的行为就像.被指定（当前目录中的包）一样；这可以与-u 标志一起使用来更新提供导入包的模块。

每个参数可能包含一个版本查询后缀(version query suffix)，指示所需的版本，例如 go get golang.org/x/text@v0.3.0。版本查询(version query)后缀由一个@符号后

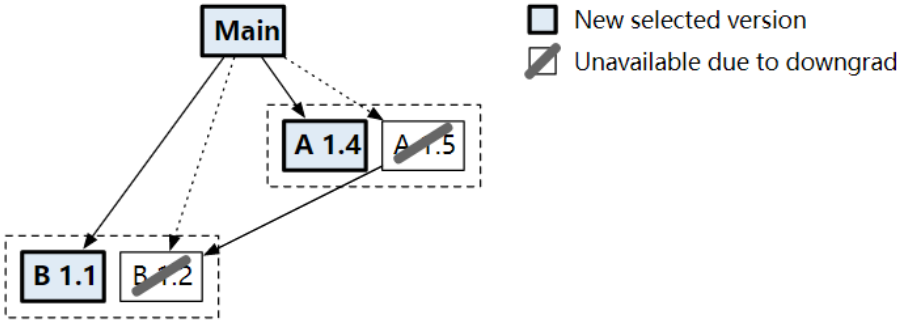
跟一个版本查询组成，它可能表示特定版本(v0.3.0)、版本前缀(v0.3)、分支或标签名称(master)、修订号(1234abcd) ，或特殊查询 latest、upgrade、patch 或 none 之一。如果没有给出版本，go get 使用@upgrade 查询。

一旦 go get 将其参数解析为特定模块和版本，go get 将在主模块的 go.mod 文件中添加、更改或删除 require 指令，以确保模块将来保持想要的版本。请注意 go.mod 文件中所需的版本是最低版本号(minimum versions)，并且可能会随着新依赖项的添加而自动增加最低版本号。有关如何选择版本以及如何通过模块感知命令解决冲突的详细信息，请参阅最小版本选择(MVS)。

如果模块的新版本需求更高版本的其他模块，则在命令行上添加、升级或降级该模块时，可能会升级其他模块。例如，假设模块 example.com/a 升级到版本 v1.5.0，并且该版本需求 v1.2.0 版本的模块 example.com/b。如果当前 v1.1.0 版本需求模块 example.com/b，则 go get example.com/a@v1.5.0 也会将 example.com/b 升级到 v1.2.0。



当在命令行中指定的模块降级或删除时，其他模块也可能会降级。继续上面的例子，假设模块 example.com/b 被降级到 v1.1.0。模块 example.com/a 也将降级为需求 example.com/b v1.1.0 或更低版本的版本。



可以使用版本后缀@none 删除模块需求。这是一种特殊的降级。依赖于已移除模块的模块将根据需要降级或移除。即使一个或多个包是由主模块中的包导入的，也可以删除一个模块需求。在这种情况下，下一个构建命令可能会添加新的模块需求。

如果两个不同版本需要一个模块（在命令行参数中明确指定或满足升级和降级），go get 将报告错误。

在 go get 选择了一组新版本之后，它会检查任何新选择的模块版本或任何在命令行给出名字的包的模块是否被收回(retracted)或弃用(deprecated)。go get 为它找到的每个撤回版本或弃用的模块打印一个警告。go list -m -u all 可用于检查所有依赖项中的撤回和弃用。

go get 更新 go.mod 文件后，它会构建在命令行中给出名字的包。可执行文件将安装在由 GOBIN 环境变量命名的目录中，如果未设置 GOPATH 环境变量，则默认为$GOPATH/bin 或$HOME/go/bin。

go get 支持以下标志：

- -d 标志告诉 go get 不要构建或安装包。使用-d 时，go get 只会管理 go.mod 中的依赖关系。不推荐使用不带-d 的 go get 来构建和安装包（从 Go 1.17 开始）。在 Go 1.18 中，-d 将始终启用。
- -u 标志告诉 go get 升级模块，它们的包被直接或间接导入，在命令行通过名字给出。-u 选择的每个模块都将升级到其最新版本，除非已经导入更高版本（预发行版）。
- -u=patch 标志（不是-u 补丁）也告诉 go get 升级依赖项，但 go get 会将每个依赖项升级到最新的补丁版本（类似于@patch 版本查询）。
- -t 标志告诉 go get 考虑构建在命令行给出名字的包的测试所需的模块。当-t 和-u 一起使用时，go get 也会更新测试依赖项。
- 不应再使用-insecure 标志。它允许 go get 解析自定义导入路径并使用不安全的方案（例如 HTTP）从存储库和模块代理中获取。GOINSECURE 环境变量提供更细粒度的控制，应改为使用它。

从 Go 1.16 开始，推荐使用 go install 命令来构建和安装程序。当与版本后缀（如@latest 或@v1.4.6）一起使用时，go install 以模块感知模式构建包，忽略当前目录或任何父目录中的 go.mod 文件（如果有的话）。

go get 更专注于管理 go.mod 中的需求。-d 标志已弃用，在 Go 1.18 中，它将始终启用。

go install

用法：

```
go install [build flags] [packages]
```

例如：

```
# 安装最新版本的程序，忽略当前目录中的 go.mod（如果有）。
$ go install golang.org/x/tools/gopls@latest

# 安装特定版本的程序。
$ go install golang.org/x/tools/gopls@v0.6.4

# 安装由当前目录中的模块选择的版本的程序。
```

```
$ go install golang.org/x/tools/gopls

# 将所有程序安装在一个目录中。
$ go install ./cmd/...
```

go install 命令构建并安装由命令行上的路径命名的包。可执行文件（main 包）安装到由 GOBIN 环境变量命名的目录中，如果没有设置 GOPATH 环境变量，则默认为$GOPATH/bin 或$HOME/go/bin。$GOROOT 中的可执行文件安装在$GOROOT/bin 或$GOTOOLDIR 而不是$GOBIN 中。不可执行的包会被构建和缓存，但不会被安装。

从 Go 1.16 开始，如果参数有版本号后缀（如@latest 或@v1.0.0），go install 以模块感知模式构建包，忽略当前目录或任何父目录中的 go.mod 文件（如果有的话）。这对于在不影响主模块依赖关系的情况下安装可执行文件很有用。

为了消除构建中使用的模块版本的歧义，参数必须满足以下约束：

- 参数必须是包路径或包模式(pattern)（带有“...”通配符）。它们不能是标准包（如 fmt）、元模式(meta-pattern)（std、cmd、all）或相对或绝对文件路径。
- 所有参数必须具有相同的版本号后缀。不允许不同的查询，即使它们指向相同的版本。
- 所有参数必须引用同一模块中相同版本的包。
- 包路径参数必须引用 main 包。模式参数只会匹配 main 包。
- 没有模块被认为是主模块(main module)。
  - 如果在命令行上给出名字的包的模块有一个 go.mod 文件，则它不能包含指令（replace 和 exclude），如果它是主模块，则会导致它被不同地解释。
  - 该模块不得依赖其自身的更高的版本。
  - vendor 目录不在任何模块中使用。（vendor 目录不包含在模块 zip 文件中，因此 go install 不会下载它们。）

有关支持的版本查询语法，请参阅 Version queries。Go 1.15 及更低版本不支持在 go install 中使用版本查询。

如果参数没有版本号后缀，则 go install 可以在模块感知模式或 GOPATH 模式下运行，具体取决于 GO111MODULE 环境变量和 go.mod 文件的存在与否。有关详细信息，请参阅 Module-aware commands。如果启用了模块感知模式，则 go install 在主模块的上下文中运行，主模块的上下文可能与包含正在被安装的包的模块（译者注：非主模块）的不同。

go list -m

用法：

```
go list -m [-u] [-retracted] [-versions] [list flags] [modules]
```

示例：

```
$ go list -m all
$ go list -m -versions example.com/m
$ go list -m -json example.com/m@latest
```

-m 标志使 go list 列出模块而不是包。在这种模式下，go list 的参数可以是模块、模块模式(pattern)（包含...通配符）、版本查询或特殊模式 all，它匹配构建列表中的所有模块。如果未指定参数，则列出主模块。

列出模块时，-f 标志仍然指定应用于 Go 结构的格式模板，但现在是 Module 结构：

```
type Module struct {
        Path       string      // module path
        Version    string      // module version
        Versions   []string    // available module versions (with -versions)
        Replace    *Module     // replaced by this module
        Time       *time.Time  // time version was created
        Update     *Module     // available update, if any (with -u)
        Main       bool        // is this the main module?
        Indirect   bool        // is this module only an indirect dependency of main module?
        Dir        string      // directory holding files for this module, if any
        GoMod      string      // path to go.mod file for this module, if any
        GoVersion  string      // go version used in module
        Deprecated string      // deprecation message, if any (with -u)
        Error      *ModuleError // error loading module
}

type ModuleError struct {
        Err string // the error itself
}
```

默认输出是打印模块路径，然后打印有关版本号和替换的信息（如果有）。例如，go list -m all 可能会打印：

```
example.com/main/module
golang.org/x/net v0.1.0
golang.org/x/text v0.3.0 => /tmp/text
rsc.io/pdf v0.1.1
```

Module 结构有一个 String 方法来格式化这行输出，因此默认格式等同于-f '{{.String}}'。

请注意，当一个模块被替换时，其 Replace 字段描述了替换模块，其 Dir 字段设置为替换模块的源代码（如果存在）。（也就是说，如果 Replace 不为空，则 Dir 设置为 Replace.Dir，无法访问被替换的源代码。）

-u 标志添加有关可用升级的信息。当给定模块的最新版本比当前版本新时，list -u 将模块的更新字段设置为有关较新模块的信息。list -u 还打印当前选择的版本是否被撤回以及模块是否被弃用。模块的 String 方法通过在当前版本号之后的括号中给出格式化的新版本号来指示可用的升级。例如，go list -m -u all 可能会打印：

```
example.com/main/module
golang.org/x/old v1.9.9 (deprecated)
golang.org/x/net v0.1.0 (retracted) [v0.2.0]
golang.org/x/text v0.3.0 [v0.4.0] => /tmp/text
rsc.io/pdf v0.1.1 [v0.1.2]
```

（对于工具软件，go list -m -u -json all 的输出可能更方便解析。）

-versions 标志使 list 将模块的 Versions 字段设置为该模块的所有已知版本的列表，根据语义版本号排序，从最低到最高。该标志还会更改默认输出格式以显示模块路径，后跟用空格分隔的版本号列表。除非还指定了-retracted 标志，否则此列表中将省略撤回的版本。

-retracted 标志指示 list 在使用-versions 标志打印的列表中显示撤回的版本，并在解析版本查询时考虑撤回的版本。例如，go list -m -retracted example.com/m@latest 显示模块 example.com/m 的最高版本或预发布版本，即使该版本已被撤回。对于此版本，从 go.mod 文件中加载 retract 指令和弃用指令。-retracted 标志是在 Go 1.16 中添加的。

模块函数 module 采用单个字符串参数，该参数必须是模块路径或查询，并将指定模块作为 Module 结构返回。如果发生错误，结果将是一个带有非空 Error 字段的 Module 结构体。

go mod download

用法：

```
go mod download [-json] [-x] [modules]
```

例如：

```
$ go mod download
$ go mod download golang.org/x/mod@v0.2.0
```

go mod download 命令将给定名字的模块下载到模块缓存中。参数可以是主模块依赖关系中的模块路径或模块模式(pattern)，也可以是 path@version 形式的版本查询。若没有参数，则 download 应用于主模块的所有依赖项。

go 命令在普通执行时会根据需要自动下载模块。go mod download 命令主要用于预填充模块缓存或加载模块代理(module proxy)服务的数据。

-json 标志导致 download 将一系列 JSON 对象打印到标准输出，描述每个下载的模块（或失败），对应于这个 Go 结构：

```
type Module struct {
        Path         string // 模块路径
        Version    string // 模块版本
        Error       string // 加载模块出错
        Info         string // 缓存的.info 文件的绝对路径
        GoMod       string // 缓存的.mod 文件的绝对路径
        Zip          string // 缓存的.zip 文件的绝对路径
        Dir          string // 缓存的源代码根目录的绝对路径
        Sum          string // 路径和版本的校验码（与 go.sum 中的一样）
        GoModSum string // go.mod 文件的校验码（与 go.sum 中的一样）
}
```

-x 标志使 download 打印 download 命令的执行信息到标准错误。

go mod edit

用法：

```
go mod edit [editing flags] [-fmt|-print|-json] [go.mod]
```

例如：

```
# 添加一个替换指令
$ go mod edit -replace example.com/a@v1.0.0=./a

# 移除一个替换指令
$ go mod edit -dropreplace example.com/a@v1.0.0

# 设置 Go 版本，添加一个依赖项，并打印输出而不是写入磁盘
$ go mod edit -go=1.14 -require=example.com/m@v1.0.0 -print

# 格式化 go.mod 文件
$ go mod edit -fmt

# 格式化并输出到另一个.mod 文件
$ go mod edit -print tools.mod

# 以 JOSN 格式输出 go.mod 文件
$ go mod edit -json
```

go mod edit 命令提供了一个用于编辑和格式化 go.mod 文件的命令行接口，主要供工具和脚本使用。go mod edit 只读取一个 go.mod 文件；它不查找有关其他模块的信息。默认情况下，go mod edit 读写主模块的 go.mod 文件，但可以在编辑标志后指定不同的目标文件。

编辑标志指定一系列编辑操作。

- -module 标志更改模块的路径（go.mod 文件的 module 行）。
- -go=version 标志设置预期的 Go 语言版本。
- -require=path@version 和-droprequire=path 标志添加和删除对给定模块路径和版本的依赖项。请注意，-require 会覆盖该路径上的任何现有依赖项。这些标志主要用于理解模块图的工具。用户应该更喜欢 go get path@version 或 go get path@none，它们会根据需要进行其他 go.mod 调整以满足其他模块施加的约束。见 go get。
- -exclude=path@version 和-dropexclude=path@version 标志添加和删除给定模块路径和版本的排除项。请注意，如果该排除项已存在，则 -exclude=path@version 不会进行任何操作。
- -replace=old[@v]=new[@v]标志添加了给定模块路径和版本号对的替换。如果 old@v 中的@v 被省略，则在左侧添加没有版本号的替换，适用于替换旧模块路径的所有版本。如果省略 new@v 中的@v，则新路径应该是本地模块根目录，而不是模块路径。请注意，-replace 会覆盖 old[@v]的任何冗余替换，因此省略@v 将删除特定版本的替换。
- -dropreplace=old[@v]标志删除给定模块路径和版本号对的替换。如果提供了@v，则删除指定版本的替换。左侧没有版本号的现有替换仍可存在，不会被删除。如果省略@v，则也会删除没有版本号的替换。
- -retract=version 和-dropretract=version 标志添加和删除给定版本的撤回，它可以是单个版本号（例如 v1.2.3）或区间（例如[v1.1.0,v1.2.0]）。请注意，-retract 标志不能为 retract 指令添加合理的注释。合理的注释，可以通过 go list -m -u 和其他命令显示。

可以重复给出编辑标志。将按给定的顺序应用这些更改。

go mod edit 有额外的标志来控制它的输出。

- -fmt 标志重新格式化 go.mod 文件而不进行其他更改。使用或重写 go.mod 文件的任何其他修改也默认也会执行这种重新格式化。唯一需要此标志的情况是没有指定其他标志，例如 go mod edit -fmt。
- -print 标志以文本格式打印最终的 go.mod，而不是将其写回磁盘。
- -json 标志以 JSON 格式打印最终的 go.mod，而不是以文本格式将其写回磁盘。JSON 输出对应于以下 Go 类型：

```
type Module struct {
        Path       string
        Version string
}

type GoMod struct {
        Module    Module
        Go          string
        Require []Require
        Exclude []Module
```

```
        Replace []Replace
}

type Require struct {
        Path        string
        Version     string
        Indirect bool
}

type Replace struct {
        Old Module
        New Module
}

type Retract struct {
        Low             string
        High            string
        Rationale string
}
```

请注意，这仅描述 go.mod 文件本身，而不是间接引用的其他模块。对于一个构建可用的完整模块集，请使用 go list -m -json all 输出。请参阅 go list -m。

例如，一个工具可以通过解析 go mod edit -json 的输出来获取 go.mod 文件的数据结构，然后可以通过使用-require、-exclude 等标志调用 go mod edit 来进行更改。

工具也可以使用 golang.org/x/mod/modfile 包来解析、编辑和格式化 go.mod 文件。

go mod graph

用法：
go mod graph [-go=version]

go mod graph 命令以文本形式打印模块需求图(module requirement graph)（应用了替换指令之后的）。例如：

```
example.com/main example.com/a@v1.1.0
example.com/main example.com/b@v1.2.0
example.com/a@v1.1.0 example.com/b@v1.1.1
example.com/a@v1.1.0 example.com/c@v1.3.0
example.com/b@v1.1.0 example.com/c@v1.1.0
example.com/b@v1.2.0 example.com/c@v1.2.0
```

模块图中的每个顶点代表模块的特定版本。图中的每条边都代表对最小版本依赖

项的需求。

go mod graph 打印图的边，每行一条。每行有两个以空格分隔的字段：一个模块版本及其依赖项之一。每个模块版本都被标识为 path@version 形式的字符串。主模块没有@version 后缀，因为它没有版本号。

-go 标志让 go mod graph 报告给定的 Go 版本加载的模块图，而不是 go.mod 文件中 go 指令指示的版本。

有关如何选择版本的更多信息，请参阅最小版本选择(MVS)。另请参阅 go list -m 打印选定版本和 go mod why 以了解为什么需要这些模块。

go mod init

用法：
go mod init [module-path]

例如：
go mod init
go mod init example.com/m

go mod init 命令在当前目录初始化并写入一个新的 go.mod 文件，实际上是在当前目录创建一个新模块。go.mod 文件必须先前不存在。

init 接受一个可选参数，即新模块的模块路径。有关选择模块路径的说明，请参阅 Module paths。如果省略了模块路径参数，init 将尝试使用.go 文件、vendor 工具配置文件和当前目录（如果在 GOPATH 中）中的导入注释来推断模块路径。

如果存在 vendor 工具的配置文件，init 将尝试从中导入模块需求。init 支持以下配置文件。

- GLOCKFILE (Glock)
- Godeps/Godeps.json (Godeps)
- Gopkg.lock (dep)
- dependencies.tsv (godeps)
- glide.lock (glide)
- vendor.conf (trash)
- vendor.yml (govend)
- vendor/manifest (gvt)
- vendor/vendor.json (govendor)

vendor 工具配置文件不能总是以完美的保真度进行翻译。例如，如果在同一个仓库中导入了多个不同版本的包，而该仓库只包含一个模块，则导入的 go.mod 只能 require 其中一个版本的模块。可以运行 go list -m all 以检查构建列表(build list)中的所有版本，运行 go mod tidy 以添加缺少的需求并删除未使用的需求。

go mod tidy

用法：
go mod tidy [-e] [-v] [-go=version] [-compat=version]

go mod tidy 确保 go.mod 文件与模块中的源代码相匹配。它添加构建当前模块的包和依赖项所需的任何缺少的模块需求，并删除对于不提供任何相关包的模块的需求。它还把所有缺失的条目添加到 go.sum 文件中并删除不必要的条目。

-e 标志（在 Go 1.16 版本添加）使 go mod tidy 尽管在加载包时遇到错误，但尝试继续执行下去。

-v 标志使 go mod tidy 将有关已删除模块的信息打印到标准错误。

go mod tidy 通过递归加载主模块中的所有包以及它们导入的所有包来工作。这包括由测试导入的包（包括其他模块中的测试）。go mod tidy 的行为就像启用了所有构建标签一样，因此它会考虑特定于平台的源文件和需要自定义构建标签的文件，即使这些源文件通常不会被构建。有一个例外：ignore 构建标记未启用，因此将不考虑具有构建约束// +build ignore 的文件。请注意，go mod tidy 不会考虑主模块中名为 testdata 的目录或名称以.或_，除非这些包由其他包显式导入。

一旦 go mod tidy 加载了这组包，它会确保每个提供一个或多个包的模块在主模块的 go.mod 文件中都有一个 require 指令，或者——如果主模块位于 go 1.16 或更低版本——还确保被另一个必需的模块所需求的模块。go mod tidy 将对每个缺少的模块的最新版本添加需求（有关 latest 版本的定义，请参阅 Version queries）。go mod tidy 将删除不提供上述集合中任何包的模块的 require 指令。

go mod tidy 还可以添加或删除对 require 指令的// indirect 注释。// indirect 注释表示这个模块不提供被主模块中的包直接导入的包。（有关何时添加// indirect 依赖项和注释的更多详细信息，请参阅 require 指令。）

如果设置了-go 标志，go mod tidy 会将 go 指令更新为指示的版本，根据该版本启用或禁用模块图化简和模块懒加载（并根据需要添加或删除间接需求）。

默认情况下，go mod tidy 将检查当模块图被 go 指令指示的 Go 版本之前的 Go 版本加载时（译者注：操作系统中安装的 Go 版本低于 go.mod 文件中 go 指令要求的 Go 版本时），模块的选定版本(selected versions)不会更改。检查版本相关的兼容性也可以通过-compat 标志显式指定。

go mod vendor

用法：

go mod vendor [-e] [-v] [-o]

go mod vendor 命令在主模块的根目录中构建一个名为 vendor 的目录，其中包含支持主模块中包的构建和测试所需的所有包的副本。不包括仅用来测试而导入的主模块外部的包。与 go mod tidy 和其他模块命令一样，构建 vendor 目录时不考虑除 ignore 之外的构建约束。

启用 vendor 模式后，go 命令将从 vendor 目录加载包，而不是将模块从其源下载到模块缓存中并使用这些下载的副本包。有关详细信息，请参阅 Vendoring。

go mod vendor 还会创建文件 vendor/modules.txt，其中包含提供的包的列表以及从它们的模块版本号。启用 vendor 模式时，此清单用作模块版本信息的来源，由 go list -m 和 go version -m 报告。go 命令读取 vendor/modules.txt 时，会检查模块版本是否与 go.mod 中的一致。如果在生成 vendor/modules.txt 后 go.mod 发生了变化，则应该再次运行 go mod vendor 命令。

请注意，如果 vendor 目录已存在，则 go mod vendor 命令会在重新构建它之前将其删除。不应对 vendor 目录的软件包进行本地更改。go 命令不会检查 vendor 目录中的包是否未被修改，但可以通过运行 go mod vendor 命令并检查是否未进行任何更改来验证 vendor 目录的完整性。

-e 标志（在 Go 1.16 中添加）让 go mod vendor 命令尝试继续执行下去，尽管在加载包时遇到错误。

-v 标志会让 go mod vendor 命令将提供的模块和包的名称打印到标准错误。

-o 标志（在 Go 1.18 中添加）让 go mod vendor 命令在指定目录而不是 vendor 目录输出提供的包的树形信息。参数可以是绝对路径或相对于模块根目录的路径。

go mod verify

用法：
go mod verify

go mod verify 检查存储在模块缓存(module cache)中的主模块的依赖关系项在下载后有没有被修改。要执行此检查，go mod verify 散列每个下载的模块.zip 文件和提取的目录，然后将这些散列值与模块首次下载时记录的散列值进行比较。go mod verify 检查构建列表中的每个模块（可以使用 go list -m all 打印）。

如果所有模块都未修改，则 go mod verify 打印“所有模块已验证(all modules verified)”。否则，它会报告哪些模块已更改并以非零状态退出。

请注意，所有模块感知命令都会验证主模块的 go.sum 文件中的散列值是否与下载到模块缓存中的模块记录的散列值匹配。如果 go.sum 中缺少某个散列值（例

如，因为第一次使用该模块），则 go 命令使用校验值数据库(checksum database)验证其散列值（除非模块路径与 GOPRIVATE 或 GONOSUMDB 环境变量匹配）。有关详细信息，请参阅验证模块(Authenticating modules)。

相比之下，go mod verify 检查模块.zip 文件及其提取的目录的散列值是否与首次下载时模块缓存中记录的散列值相匹配。这对于在下载和验证一个模块后检测模块缓存中其文件是否更改很有用。go mod verify 不会为不在缓存中的模块下载内容，也不会使用 go.sum 文件来验证模块内容。但是，go mod verify 可能会下载 go.mod 文件以执行最小版本选择算法。它将使用 go.sum 来验证这些文件，并且它可能会把丢失的散列值添加到 go.sum 条目。

go mod why

用法：
go mod why [-m] [-vendor] packages...

go mod why 根据依赖图显示从主模块到每个列出的包的最短路径。

输出一系列节，每一个节用于命令行上给出的每个包或模块，由空行分隔。每个节都以注释行开头，以#开头，给出目标包或模块。其后的行给出了根据依赖图推导出的路径，每行一个包。如果未从主模块中引用该包或模块，则该节将显示一个带括号的注释来指出该事实。

例如：
```
$ go mod why golang.org/x/text/language golang.org/x/text/encoding
# golang.org/x/text/language
rsc.io/quote
rsc.io/sampler
golang.org/x/text/language

# golang.org/x/text/encoding
(main module does not need package golang.org/x/text/encoding)
```

-m 标志让 go mod why 将其参数视为模块列表。go mod why 会打印每个模块中所有包的路径。请注意，即使使用了-m 标志，go mod why 也会去查询包依赖图，而不是 go mod graph 的打印输出。

-vendor 标志让 go mod why 忽略主模块之外的测试用导入的包（就像 go mod vendor 所做的那样）。默认情况下，go mod why 会考虑与 all 模式匹配的包依赖图。在 go.mod 中使用 go 指令声明了 go 1.16 或更高版本的模块，在 Go 1.16 之后的构建环境中此标志无效，因为 all 的含义已更改为匹配 go mod vendor 匹配的包集。

go version -m

用法：
go version [-m] [-v] [file ...]

例如：
```
# 打印用于构建 go 项目的 Go 版本。
$ go version

# 打印构建一个指定可执行文件的 Go 版本。
$ go version ~/go/bin/gopls

# 打印构建一个指定可执行文件的 Go 版本和模块版本。
$ go version -m ~/go/bin/gopls

# 打印用来构建一个指定目录中的可执行文件的 Go 版本和模块版本。
$ go version -m ~/go/bin/
```

go version 报告用来构建在命令行上给出名字的每个可执行文件的 Go 版本。

如果在命令行上没有给出名字的文件，go version 会打印自己的版本信息。

如果在命令行上给出一个目录，go version 会递归地遍历该目录，寻找可识别的 Go 二进制文件并报告它们的版本。默认情况下，go version 不报告在目录扫描期间发现的无法识别的文件。-v 标志让它报告无法识别的文件。

-m 标志使 go version 打印每个可执行文件的嵌入的模块版本信息（如果可用）。对于每个可执行文件，go version -m 打印一个带有制表符分隔列的表格，如下所示。
```
$ go version -m ~/go/bin/goimports
/home/jrgopher/go/bin/goimports: go1.14.3
        path    golang.org/x/tools/cmd/goimports
        mod     golang.org/x/tools      v0.0.0-20200518203908-8018eb2c26ba
h1:0Lcy64USfQQL6GAJma8BdHCgeofcchQj+Z7j0SXYAzU=
        dep     golang.org/x/mod        v0.2.0
h1:KU7oHjnv3XNWfa5COkzUifxZmxp1TyI7ImMXqFxLwvQ=
        dep     golang.org/x/xerrors    v0.0.0-20191204190536-9bdfabe68543
h1:E7g+9GITq07hpfrRu66IVDexMakfv52eLZ2CXBWiKr4=
```

表格的格式将来可能会改变。可以从 runtime/debug.ReadBuildInfo 获得相同的信息。

表中每一行的含义由第一列中的单词决定。

- path：用于构建可执行文件的 main 包的路径。

- mod：包含 main 包的模块。这些列分别是模块路径、版本号和散列值。主模块有版本号(devel)，但没有散列值。
- dep：提供一个或多个链接到可执行文件的包的模块。格式与 mod 相同。
- =>：上一行的模块的一个替换(replacement)。如果这个替换是本地的目录，则仅列出目录路径（无版本号或散列值）。如果这个替换是模块的版本，则列出路径、版本号和散列值，格式与 mod 和 dep 一样。被替换的模块没有散列值。

go clean -modcache

用法：
go clean [-modcache]

-modcache 标志让 go clean 删除整个模块缓存，包括有版本号的依赖项对应的未打包的源代码。

这通常是删除模块缓存的最佳方式。默认情况下，模块缓存中的大多数文件和目录都是只读的，以防止测试和编辑器在经过身份验证(authenticated)后无意间更改这些文件。不幸的是，这会导致像 rm -r 这样的命令执行失败，因为如果不先使它们的父目录可写，就无法删除这些文件。

-modcacherw 标志（被 go build 和其他模块感知命令接受）让模块缓存中的新目录可写。要将-modcacherw 传递给所有模块感知命令，请将其添加到 GOFLAGS 环境变量。GOFLAGS 可以在环境中设置或使用 go env -w 命令设置。例如，下面的命令将其永久设置：
go env -w GOFLAGS=-modcacherw

-modcacherw 应谨慎使用；开发人员应注意不要更改模块缓存中的文件。go mod verify 可用于检查缓存中的文件是否与主模块的 go.sum 文件中的散列值匹配。

版本查询(Version query)

好几个命令允许你使用版本查询指定模块的版本，版本查询出现在命令行上模块或包路径后面的@字符之后。

例如：
go get example.com/m@latest
go mod download example.com/m@master
go list -m -json example.com/m@e3702bed2

版本查询可能是以下之一：
- 完全指定的语义化版本号，例如 v1.2.3，它选择特定版本。有关语法，请参阅版本号(Versions)。

- 语义化版本号前缀，例如 v1 或 v1.2，它选择具有该前缀的最高可用版本。
- 语义化版本号比较，例如<v1.2.3 或>=v1.5.6，它选择最接近比较目标的可用版本（>和>=的最低版本，<和<=的最高版本）。
- 源代码存储库隐含的修订标识符，例如 commit 哈希前缀、修订标签或分支名称。如果修订标签就是语义化版本号，则此查询选择该版本。否则，此查询为隐含的 commit 选择一个伪版本号(pseudo-version)。请注意，不能以这种方式选择名称与其他版本查询匹配的分支和标签。例如，查询 v2 选择以 v2 开头的最新版本，而不是名为 v2 的分支。
- 字符串 latest，它选择最高可用的发行版本。如果没有发行版本，latest 选择最高的预发布版本。如果没有打了标签的版本，latest 会在存储库默认分支的最新 commit 选择一个伪版本号。
- 字符串 upgrade，与 latest 类似，但如果当前需求的模块版本高于 latest 将选择的版本（例如，预发布版本），则 upgrade 将选择当前版本。
- 字符串 patch，它选择与当前所需版本具有相同主版本号和次版本号的最新可用版本。如果当前还没有需求的版本，则 patch 相当于 latest。从 Go 1.16 开始，go get 在使用 patch 时需求当前版本（但-u=patch 标志没有此需求）。

除了对特定给出名称的版本或修订的查询外，所有查询都与 go list -m -versions 报告的可用版本（请参阅 go list -m）相关。此列表仅包含加了标签的版本号，不包含伪版本号。不考虑主模块的 go.mod 文件中的 exclude 指令不允许的模块版本。来自同一模块的最新版本的 go.mod 文件中被 retract 指令覆盖的版本也被忽略，除非-retracted 标志与 go list -m 一起使用并且加载 retract 指令时除外。

发布版本优先于预发布版本。例如，如果版本 v1.2.2 和 v1.2.3-pre 可用，则 latest 查询将选择 v1.2.2，即使 v1.2.3-pre 更高。<v1.2.4 查询也会选择 v1.2.2，即使 v1.2.3-pre 更接近 v1.2.4。如果没有可用的发布或预发布版本，则 latest、upgrade 和 patch 查询将为存储库默认分支的最新的 commit 选择一个伪版本号。其他查询会报错。

在模块外部执行模块命令(Module commands outside a module)

模块感知的 Go 命令通常在由工作目录或父目录中的 go.mod 文件定义的主模块的上下文中运行。一些命令可以在没有 go.mod 文件的情况下以模块感知模式运行，但大多数命令的工作方式与此不同，或者在没有 go.mod 文件时报告一个错误。

有关启用和禁用模块感知模式的信息，请参阅模块感知命令(Module-aware commands)。

命令	行为
go build	
go doc	
go fix	
go fmt	

go generate	只有标准库中的包和命令行中指定.go文件的包才能被加载、导入和构建。无法构建来自其他模块的包，因为没有地方记录模块需求并确保确定性构建。
go install	
go list	
go run	
go test	
go vet	
go get	包和可执行文件可以像往常一样构建和安装。请注意，在没有 go.mod 文件的情况下运行 go get 时没有主模块，因此不应用 replace 和 exclude 指令。
go list -m	大多数参数都需要显式的版本查询，除非使用-versions 标志。
go mod download	大多数参数都需要显式的版本查询。
go mod edit	需要显式的文件参数。
go mod graph	
go mod tidy	
go mod vendor	这些命令需要一个 go.mod 文件，如果不存在将报告错误。
go mod verify	
go mod why	

go work init

用法：
go work init [moddirs]

该命令在当前目录中初始化并写入一个新的 go.work 文件，作用是在当前目录中创建一个新的工作空间。

go work init 可接受工作区模块的路径作为参数。如果省略该参数，将创建一个没有模块的空工作区。

每个参数路径都添加到 go.work 文件中的 use 指令中。当前的 go 版本也将列在 go.work 文件中。

go work edit

用法：
go work edit [editing flags] [go.work]

go work edit 命令提供了一个用于编辑 go.work 的命令行接口，主要给工具或脚本使用。它只读取 go.work，它不查找所涉及模块的信息。如果没有指定文件，该命令在当前目录及其父目录中查找 go.work 文件

editing flags 指定一系列编辑操作。

- -fmt 标志重新格式化 go.work 文件而不进行其他更改。使用或重写 go.work 文件的任何其他修改也隐含了这种重新格式化。唯一需要此标志的情况是没有指定其他标志，例如“go work edit -fmt”。
- -use=path 和-dropuse=path 标志从 go.work 文件的模块目录集中添加和删除 use 指令。
- -replace=old[@v]=new[@v]标志添加了给定模块路径和版本对的替换指令。如果 old@v 中的@v 被省略，则添加左侧没有版本号的替换指令，适用于旧模块路径的所有版本。如果省略 new@v 中的@v，则新路径应该是本地模块的根目录，而不是模块路径。请注意，-replace 会覆盖 old[@v]的任何已有替换指令，因此省略@v 将删除特定版本的现有替换指令。
- -dropreplace=old[@v]标志删除给定模块路径和版本对的替换指令。如果省略了@v，则会删除左侧没有版本的替换指令。
- -go=version 标志设置想要的 Go 语言版本。

可以重复给出 editing flags 标志，将按给定的顺序执行更改。

go work edit 有额外的标志来控制它的输出：

- -print 标志以文本格式打印输出最终的 go.work，而不是将其写回 go.work。
- -json 标志以 JSON 格式打印输出最终的 go.work，而不是将其写回 go.work。JSON 输出对应于以下 Go 类型：

```
type Module struct {
	Path	string
	Version string
}

type GoWork struct {
	Go		string
	Directory []Directory
	Replace	[]Replace
}

type Use struct {
	Path		string
	ModulePath string
}

type Replace struct {
	Old Module
```

```
    New Module
}
```

go work use

用法：
```
go work use [-r] [moddirs]
```

`go work use` 命令提供了一个命令行接口，用于将目录（可以以递归方式）添加到 `go.work` 文件中。

命令行上列出的每个参数目录被 use 指令添加到 `go.work` 文件中（如果它们存在于磁盘上），或者从 `go.work` 文件中删除（如果它们不存在于磁盘上）。

`-r` 标志递归搜索参数目录中的模块，并且 `use` 命令就像每个目录都被指定为命令行参数一样操作：在磁盘上存在的目录添加到 `go.work` 文件中的 `use` 指令，在磁盘上不存在的目录则删除对应 `use` 指令。

go work sync

用法：
```
go work sync
```

`go work sync` 命令将工作区的构建列表同步回工作区的模块。

工作区的构建列表用来在工作区中（传递地）构建所有依赖模块的版本集。`go work sync` 使用最小版本选择(MVS)算法生成该构建列表，然后将这些版本同步回工作区中指定的每个模块（使用 `use` 指令）。

一旦计算了工作区的构建列表，工作区中记录每个模块的 `go.mod` 文件会被重写，并使用与该模块相关的依赖项进行升级以匹配工作区构建列表。请注意，最小版本选择算法保证每个模块在构建列表中的版本始终与在工作区中的版本相同或更高。

模块代理(Module proxies)

GOPROXY 协议(GOPROXY protocol)

模块代理是一个 HTTP 服务器，它可以响应下面指定的路径的 `GET` 请求。这些请求没有查询参数，也不需要特定的请求头字段，因此即使是从固定文件系统（包括 `file://URL`）提供服务的站点也可以是模块代理。

成功的 HTTP 响应必须具有状态代码 200(OK)。重定向(3xx)会被遵循。状态码为 4xx 和 5xx 的响应被视为错误。错误代码 404（未找到）和 410（已消失）表示请求的模块或版本在代理上不可用，但可以在其他地方找到。错误响应的内容类型应该是 text/plain，字符集为 utf-8 或 us-ascii。

go 命令可以配置使用 GOPROXY 环境变量联系代理或源代码控制服务器，该变量接受代理 URL 列表。该列表可能包含关键字 direct 或 off（有关详细信息，请参阅环境变量(Environment variables)）。列表元素可以用逗号(,)或竖线(|)分隔，这决定了遇到错误时的回退行为。当 URL 后跟逗号时，go 命令仅在 404（未找到）或 410（已消失）响应后回退到后面的源。当 URL 后跟管道时，go 命令会在出现任何错误（包括超时等非 HTTP 错误）后回退到后面的源。这种错误处理行为让代理充当未知模块的看门人。例如，对于不在合法认证的列表中的模块，代理可能会以错误 403（禁止）来响应（请参阅为私有模块提供服务的私有代理(Private proxy serving private modules)）。

下表指定了模块代理必须响应的查询方式。对于每个路径，$base 是代理 URL 的路径部分，$module 是模块路径，$version 是版本。例如，如果代理 URL 是 https://example.com/mod，并且客户端正在为版本 v0.3.2 的模块 golang.org/x/text 请求 go.mod 文件，则客户端将发送一个 GET 请求 https://example.com/mod/golang.org/x/text/@v/v0.3.2.mod。

为了避免在不区分大小写的文件系统中提供服务时产生歧义，$module 和 $version 元素通过将每个大写字母替换为感叹号后跟相应的小写字母来进行编码。这允许模块 example.com/M 和 example.com/m 都存储在磁盘上，因为前者被编码为 example.com/!m。

路径	描述

$base/$module/@v/list	以纯文本形式返回给定模块的已知版本列表，每行一个。此列表不应包括伪版本。
	
	返回有关模块特定版本的 JSON 格式的元数据。响应必须是与下面的 Go 数据结构相对应的 JSON 对象：
	type Info struct {
	Version string // version string
	Time time.Time // commit time
	}

$base/$module/@v/$version.info	Version 字段是必需的，并且必须包含有效的规范版本号(canonical version)（请参阅版本号(Versions)）。请求路径中的$version 不需要是相匹配的版本号，甚至不需要是有效的版本号；此端点可用于查找分支名版本号称或修订

	标识符版本号。但是，如果$version 是与$module 兼容的主版本号的规范版本号，则成功响应中的 Version 字段必须与此相同。 Time 字段是可选的。如果存在，它必须是 RFC 3339 格式的字符串。它表示版本的创建时间。 未来可能会添加更多字段，因此保留其他名称。
$base/$module/@v/$version.mod	返回模块特定版本的 go.mod 文件。如果模块没有请求版本的 go.mod 文件，则必须返回 module 声明语句仅包含请求的模块路径的 go.mod 文件。否则，必须返回原始的、未修改的 go.mod 文件。
$base/$module/@v/$version.zip	返回一个包含特定版本模块内容的 zip 文件。有关必须如何格式化此 zip 文件的详细信息，请参阅模块 zip 文件 (Module zip files)。
$base/$module/@latest	以与 $base/$module/@v/$version.info 相同的格式返回有关模块的最新已知版本的 JSON 格式元数据。如果 $base/$module/@v/list 为空或没有适合的版本列出，则最新版本应该是 go 命令使用的模块版本。此端点是可选的，不需要模块代理来实现它。

在解析模块的最新版本时，go 命令会请求$base/$module/@v/list，如果没有找到合适的版本，则请求$base/$module/@latest。go 命令按顺序优先选择：语义上最高的发布版本号、语义上最高的预发布版本号和按时间顺序排列的最新伪版本号。在 Go 1.12 及更早版本中，go 命令将$base/$module/@v/list 中的伪版本号视为预发布版本号，但自 Go 1.13 起不再如此。

模块代理必须始终为$base/$module/$version.mod 和$base/$module/$version.zip 查询的成功响应提供相同的内容。此内容使用 go.sum 文件进行加密验证 (cryptographically authenticated)，默认情况下使用校验值数据库 (checksum database)。

go 命令将从模块代理下载的大部分内容缓存在 $GOPATH/pkg/mod/cache/download 的模块缓存中。即使直接从版本控制系统下载，go 命令也会合成显式的 info、mod 和 zip 文件并将它们存储在此目录中，就像它直接从代理下载它们一样。缓存的布局与代理 URL 空间相同，因此在

https://example.com/proxy 提供（或将其复制到）$GOPATH/pkg/mod/cache/download 将允许用户通过设置 GOPROXY 到 https://example.com/proxy 访问缓存的模块版本 。

与代理通信(Communicating with proxies)

go 命令可以从模块代理下载模块源代码和元数据。GOPROXY 环境变量可用于配置 go 命令可以连接到哪些代理以及它是否可以直接与版本控制系统通信。下载的模块数据保存在模块缓存中。go 命令只会在需要的信息在模块缓存中不存在时联系代理。

GOPROXY 协议部分描述了可以发送到 GOPROXY 服务器的请求。但是，了解 go 命令何时发出这些请求也很有帮助。例如，go build 遵循以下过程：

- 通过读取 go.mod 文件并执行最小版本选择(MVS)算法来计算构建列表。
- 阅读命令行中给出名字的包以及它们导入的包。
- 如果构建列表中的任何模块都没有提供某个包，那么会去找到提供它的模块，并把对该包的最新版本的需求添加到 go.mod 文件里，然后重新开始。
- 加载完所有内容后构建包。

当 go 命令计算构建列表时，它会为模块图中的每个模块加载 go.mod 文件。如果 go.mod 文件不在缓存中，go 命令将使用$module/@v/$version.mod 请求从代理下载它（其中$module 是模块路径，$version 是版本号）。这些请求可以使用 curl 之类的工具进行测试。例如，下面的命令会下载版本为 v0.2.0 的 golang.org/x/mod 的 go.mod 文件：

```
$ curl https://proxy.golang.org/golang.org/x/mod/@v/v0.2.0.mod
module golang.org/x/mod

go 1.12

require (
	golang.org/x/crypto v0.0.0-20191011191535-87dc89f01550
	golang.org/x/tools v0.0.0-20191119224855-298f0cb1881e
	golang.org/x/xerrors v0.0.0-20191011141410-1b5146add898
)
```

为了加载一个包，go 命令需要提供它的模块的源代码。模块源代码以.zip 文件的形式分发，这些文件被提取到模块缓存中。如果模块.zip 文件不在缓存中，go 命令将使用$module/@v/$version.zip 请求下载它。

```
$ curl -O https://proxy.golang.org/golang.org/x/mod/@v/v0.2.0.zip
$ unzip -l v0.2.0.zip | head
Archive:  v0.2.0.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
     1479  00-00-1980 00:00   golang.org/x/mod@v0.2.0/LICENSE
```

1303	00-00-1980 00:00	golang.org/x/mod@v0.2.0/PATENTS
559	00-00-1980 00:00	golang.org/x/mod@v0.2.0/README
21	00-00-1980 00:00	golang.org/x/mod@v0.2.0/codereview.cfg
214	00-00-1980 00:00	golang.org/x/mod@v0.2.0/go.mod
1476	00-00-1980 00:00	golang.org/x/mod@v0.2.0/go.sum
5224	00-00-1980 00:00	golang.org/x/mod@v0.2.0/gosumcheck/main.go

请注意，.mod 和.zip 请求是分开的，即使 go.mod 文件通常包含在.zip 文件中。go 命令可能需要为许多不同的模块下载 go.mod 文件，而.mod 文件比.zip 文件小得多。此外，如果一个 Go 项目没有 go.mod 文件，则代理将生成并提供一个仅包含 module 指令的 go.mod 文件。从版本控制系统下载时，go 命令会生成一个 go.mod 文件。

如果 go 命令需要加载构建列表中没有任何模块提供的包，它将尝试去找到提供它的新模块。将包解析为模块(Resolving a package to a module)一节描述了这个过程。总之，go 命令请求可能包含该包的每个模块路径的最新版本的有关信息。例如，对于包 golang.org/x/net/html，go 命令会尝试去查找 golang.org/x/net/html、golang.org/x/net 、golang.org/x/ 和 golang.org 这些模块的最新版本。只有 golang.org/x/net 实际存在并提供该包，go 命令才使用该模块的最新版本。如果多个模块都提供该包，go 命令将使用路径最长的那个模块。

当 go 命令请求一个模块的最新版本时，它首先发送一个对$module/@v/list 的请求。如果列表为空或返回的版本都不能使用，它会发送$module/@latest 请求。选择版本后，go 命令会发送$module/@v/$version.info 请求元数据。然后它可能会发送$module/@v/$version.mod 和$module/@v/$version.zip 请求以加载 go.mod 文件和源代码。

```
$ curl https://proxy.golang.org/golang.org/x/mod/@v/list
v0.1.0
v0.2.0

$ curl https://proxy.golang.org/golang.org/x/mod/@v/v0.2.0.info
{"Version":"v0.2.0","Time":"2020-01-02T17:33:45Z"}
```

下载.mod 或.zip 文件后，go 命令计算散列值并检查它是否与主模块的 go.sum 文件中的散列值匹配。如果在 go.sum 中还不存在散列值，默认情况下，go 命令会去校验值数据库(checksum database)中检索它。如果计算的散列值不匹配，go 命令会报告安全错误并且不会将该文件安装到模块缓存中。GOPRIVATE 和 GONOSUMDB 环境变量可用于禁用对特定模块的校验值数据库的请求。GOSUMDB 环境变量也可以设置为 off 以完全禁用对校验值数据库的请求。有关更多信息，请参阅验证模块(Authenticating modules)。请注意，请求.info 返回的版本列表和版本元数据信息未经过身份验证，并且可能会随时间而变化。

直接从代理获得模块服务(Serving modules directly from a proxy)

大多数模块都是使用版本控制系统的存储库开发和提供的。在直接模式(direct mode)下，go 命令使用版本控制工具下载此类模块（请参阅版本控制系统(Version control systems)）。也可以直接从模块代理提供模块服务。这对于希望在不暴露版本控制系统的服务器的情况下提供模块服务以及使用 go 命令不支持的版本控制工具时非常有用。

当 go 命令以直接模式下载模块时，它首先根据模块路径使用 HTTP GET 请求查找模块服务器的 URL。它在 HTML 响应中查找名称为 go-import 的<meta>标签。标签的内容必须包含存储库根路径(repository root path)、版本控制系统和 URL，以空格分隔。有关详细信息，请参阅从模块路径查找存储库(Finding a repository for a module path)。

如果版本控制系统是 mod，go 命令使用 GOPROXY 协议从给定的 URL 下载模块。

例如，假设 go 命令正在尝试下载 v1.0.0 版本的模块 example.com/gopher。它向 https://example.com/gopher?go-get=1 发送请求。服务器使用包含标签的 HTML 文档进行响应：

```
<meta name="go-import" content="example.com/gopher mod https://modproxy.example.com">
```

基于此响应，go 命令通过发送对 https://modproxy.example.com/example.com/gopher/@v/v1.0.0.info、v1.0.0.mod 和 v1.0.0.zip 的请求来下载模块。

请注意，直接从代理提供的模块服务无法在 GOPATH 模式下使用 go get 命令下载。

版本控制系统(Version control systems)

go 命令可以直接从版本控制系统的存储库下载模块源代码和元数据。从代理下载模块通常更快，但如果代理不可用或代理无法访问模块的存储库（私有存储库通常如此），则需要直接连接到存储库。支持 Git、Subversion、Mercurial、Bazaar 和 Fossil。版本控制工具必须安装在 PATH 环境变量中的目录中，以便 go 命令使用它。

要从源存储库而不是代理下载特定模块，请设置 GOPRIVATE 或 GONOPROXY 环境变量。要将 go 命令配置为直接从源存储库下载所有模块，请将 GOPROXY 设置为 direct。有关详细信息，请参阅环境变量(Environment variables)。

从模块路径查找存储库(Finding a repository for a module path)

当 go 命令以 direct 模式下载模块时，它首先定位包含该模块的存储库。

如果模块路径在路径组件的末尾有 VCS 限定符（.bzr、.fossil、.git、.hg、.svn 之一），则 go 命令将使用该路径限定符之前的所有内容作为存储库 URL。例如，对于模块 example.com/foo.git/bar，go 命令使用 git 下载位于 example.com/foo.git 的存储库，期望在 bar 子目录中找到该模块。go 命令会根据版本控制工具支持的协议来猜测要使用的协议。

如果模块路径没有限定符，则 go 命令将 HTTP GET 请求发送到从模块路径派生的 URL，并带有?go-get=1 查询字符串。例如，对于模块 golang.org/x/mod，go 命令可能会发送以下请求：
https://golang.org/x/mod?go-get=1 (preferred)
http://golang.org/x/mod?go-get=1 (fallback, only with GOINSECURE)

go 命令遵循重定向但忽略响应状态码，因此服务器可能会以 404 或任何其他错误状态响应。GOINSECURE 环境变量可以设置为允许回退并重定向到特定模块的未加密 HTTP。

服务器必须响应一个 HTML 文档，该文档的<head>中包含<meta>标签。<meta>标签应该出现在文档的前面，以避免 go 命令的受限的解析器混淆。特别是，它应该出现在任何原始 JavaScript 或 CSS 之前。<meta>标签必须具有以下形式：
<meta name="go-import" content="root-path vcs repo-url">

root-path 是存储库根路径，即模块路径中对应于存储库根目录的部分。它必须是请求的模块路径的前缀或完全匹配。如果不是完全匹配，则会对前缀发出另一个请求以验证<meta>标签是否匹配。

vcs 是版本控制系统。它必须是下表中列出的工具之一或关键字 mod，它指示 go 命令使用 GOPROXY 协议从给定 URL 下载模块。有关详细信息，请参阅直接从代理获得模块服务(Serving modules directly from a proxy)。

repo-url 是存储库的 URL。如果 URL 不包含方案(scheme)（因为模块路径具有 VCS 限定符或因为<meta>标签中缺少方案），go 命令将尝试版本控制系统支持的每个协议。例如，对于 Git，go 命令会先尝试 https://，然后再尝试 git+ssh://。仅当模块路径与 GOINSECURE 环境变量匹配时，才能使用不安全的协议（如 http://和 git://）。

名称	命令	GOVCS 默认值	安全方案
Bazaar	bzr	Private only	https, bzr+ssh
Fossil	fossil	Private only	https
Git	git	Public and private	https, git+ssh, ssh
Mercurial	hg	Public and private	https, ssh

Subversion	svn	Private only	https, svn+ssh

例如，再次考虑 golang.org/x/mod。go 命令向 https://golang.org/x/mod?go-get=1 发送请求。服务器使用包含标签的 HTML 文档进行响应：

```
<meta name="go-import" content="golang.org/x/mod git https://go.googlesource.com/mod">
```

从这个响应中，go 命令将使用位于远程 URL https://go.googlesource.com/mod 的 Git 存储库。

GitHub 和其他流行的托管服务响应?go-get=1 对所有存储库的查询，因此通常不需要为这些站点上托管的模块进行服务器配置。

找到存储库 URL 后，go 命令会将存储库克隆到模块缓存中。通常，go 命令会尽量避免从存储库中获取不需要的数据。但是，实际使用的命令因版本控制系统而异，并且可能随时间而变化。对于 Git，go 命令可以列出大多数可用版本，而无需下载 commit。它通常会在不下载祖先 commit 的情况下获取提交，但有时有必要下载祖先 commit。

版本号映射到 commit(Mapping versions to commits)

go 命令可以在特定的规范版本号（例如 v1.2.3、v2.4.0-beta 或 v3.0.0+incompatible）上签出存储库中的模块。每个模块版本在存储库中都应该有一个语义版本号标签，指示应该为给定版本签出哪个修订版。

如果模块定义在存储库根目录或根目录的主版本号子目录中，则每个版本标签名称都等于相应的版本号。例如，模块 golang.org/x/text 定义在其存储库的根目录中，因此版本 v0.3.2 在该存储库中具有标签 v0.3.2。大多数模块都是如此。

如果一个模块定义在存储库内的子目录中，即模块路径的模块子目录部分不为空，则每个标签名称必须以模块子目录名为前缀，后跟一个斜杠。例如，模块 golang.org/x/tools/gopls 定义在存储库的 gopls 子目录中，根路径为 golang.org/x/tools。该模块的 v0.4.0 版本必须在该存储库中具有名为 gopls/v0.4.0 的标签。

语义版本号标签的主版本号必须与模块路径的主版本号后缀（如果有）一致。例如，标签 v1.0.0 可能属于模块 example.com/mod，但不属于 example.com/mod/v2，后者具有类似 v2.0.0 的标签。

如果不存在 go.mod 文件，并且该模块位于存储库根目录中，则具有主版本号 v2 或更高版本的标签可能属于没有主版本号后缀的模块。这种版本号用后缀 +incompatible 表示。版本号标签本身不能有后缀。请参阅与非模块存储库的兼容性(Compatibility with non-module repositories)。

创建标签后，不应将其删除或更改为不同的修订版本号。版本号经过认证以确保安全、可重复的构建。如果一个标签被修改，客户端在下载它时可能会看到一个安全错误。即使标签被删除，它的内容也可能在模块代理上仍然可用。

伪版本号映射到 commit(Mapping pseudo-versions to commits)

go 命令可以检查存储库中特定版本的模块，编码为伪版本号(pseudo-version)，例如 v1.3.2-0.20191109021931-daa7c04131f5。

伪版本号的最后 12 个字符（上例中的 daa7c04131f5）表示要签出的存储库中的修订号。其含义取决于版本控制系统。对于 Git 和 Mercurial，这是 commit 的散列值的前缀。对于 Subversion，这是一个用 0 填充的修订号。

在签出 commit 之前，go 命令会验证时间戳（上面的 20191109021931）是否与 commit 日期匹配。它还验证基本版本号（v1.3.1，上例中的 v1.3.2 之前的版本号）是否对应于作为 commit 祖先的语义版本标签。这些检查确保模块作者可以完全控制伪版本与其他已发布版本的比较。

有关详细信息，请参阅伪版本号(Pseudo-versions)。

分支和 commit 映射到版本号(Mapping branches and commits to versions)

可以使用版本查询(version query)在特定分支、标签或修订中签出模块。
go get example.com/mod@master

go 命令将这些名称转换为可用于最小版本选择(MVS)的规范版本号(canonical versions)。MVS 依赖于明确地排序版本的能力。随着时间的推移，分支名称和修订版本号无法可靠地进行比较，因为它们依赖于可能发生变化的存储库结构。

如果修订版带有一个或多个语义版本号标签（如 v1.2.3），则将使用最高有效版本号的标签。go 命令只考虑可能属于目标模块的语义版本号标签；例如 example.com/mod/v2 不会考虑标签 v1.5.2，因为主版本号与模块路径的后缀不匹配。

如果修订版没有使用有效的语义版本号标签进行标记，go 命令将生成一个伪版本号。如果修订版有具有有效语义版本号标签的祖先 commit，则最高祖先版本号将用作伪版本号的基础。请参阅伪版本号(Pseudo-versions)。

仓库里的模块目录(Module directories within a repository)

一旦模块的存储库在特定版本中被签出，go 命令必须找到包含模块的 go.mod 文件的目录（模块的根目录）。

回想一下，模块路径由三部分组成：存储库根路径（对应于存储库根目录）、模块子目录和主版本号后缀（仅适用于 v2 或更高版本的模块）。

对于大多数模块来说，模块路径等于存储库根路径，因此模块的根目录就是存储库的根目录。

模块有时在存储库的子目录中定义。这通常适用于具有多个需要独立发布和版本控制的组件的大型存储库。这样的模块应该在与存储库根路径之后的模块路径部分匹配的子目录中找到。例如，假设模块 example.com/monorepo/foo/bar 位于具有根路径 example.com/monorepo 的存储库中。它的 go.mod 文件必须在 foo/bar 子目录中。

如果一个模块以主版本号 v2 或更高版本发布，它的路径必须有一个主版本号后缀。带有主版本号后缀的模块可以定义在两个子目录之一中：一个有后缀，一个没有后缀。例如，假设上述模块的新版本发布了，路径为 example.com/monorepo/foo/bar/v2。它的 go.mod 文件可能在 foo/bar 或 foo/bar/v2 中。

带有主版本号后缀的子目录是“主版本子目录”。它们可用于在单个分支上开发模块的多个主版本。当多个主版本的开发在不同的分支上进行时，这可能是不必要的。但是，主版本子目录有一个重要的属性：在 GOPATH 模式下，包导入路径与 GOPATH/src 下的目录完全匹配。go 命令在 GOPATH 模式下提供最小模块兼容性（请参阅与非模块存储库的兼容性(Compatibility with non-module repositories)），因此主版本子目录并不总是与在 GOPATH 模式下构建的项目兼容。不过，不支持最小模块兼容性的旧工具可能会出现问题。

一旦 go 命令找到模块根目录，它会创建一个包含目录内容的.zip 文件，然后将.zip 文件解压缩到模块缓存中。有关.zip 文件中可能包含哪些文件的详细信息，请参阅文件路径和大小限制(File path and size constraints)。.zip 文件的内容在提取到模块缓存之前经过验证(authenticated)，就像从代理下载.zip 文件时一样。

模块 zip 文件不包含 vendor 目录或任何嵌套模块（包含 go.mod 文件的子目录）的内容。这意味着模块必须注意不要引用其目录之外或其他模块中的文件。例如，//go:embed 模式不能匹配嵌套模块中的文件。在文件不应该包含在模块中的情况下，此行为可能是一种有用的解决方案。例如，如果存储库有大文件检入 testdata 目录，模块作者可以在 testdata 中添加一个空的 go.mod 文件，这样他们的用户就不需要下载这些文件了。当然，这可能会减少用户测试其依赖项的覆盖率。

LICENSE 文件的特例(Special case for LICENSE files)

当 go 命令为不位于存储库根目录中的模块创建.zip 文件时，如果该模块在其根目录中没有名为 LICENSE 的文件（与 go.mod 放在一起），则 go 命令将复制名为 LICENSE 的文件，如果它存在于同一修订版的存储库根目录中的话。

这种特殊情况允许相同的 LICENSE 文件应用于存储库中的所有模块。这仅适用于名为 LICENSE 的特殊文件，没有.txt 之类的扩展名。不幸的是，如果不破坏现有模块的加密校验值（译者注：散列值），就无法扩展它；请参阅验证模块(Authenticating modules)。其他工具和网站（如 pkg.go.dev）可能会识别具有其他名称的文件。

另请注意，go 命令在创建模块.zip 文件时不包含符号链接；请参阅文件路径和大小限制(File path and size constraints)。因此，如果存储库的根目录中没有 LICENSE 文件，则作者可以改为在子目录中定义的模块中创建其许可证文件的副本，以确保这些文件包含在模块.zip 文件中。

使用 GOVCS 环境变量控制版本控制工具(Controlling version control tools with GOVCS)

go 命令下载带有 git 等版本控制命令的模块的能力对于去中心化包生态系统至关重要，代码可以从任何服务器导入。如果恶意服务器找到一种方法使调用的版本控制命令运行意外代码，这是一个潜在的安全问题。

为了平衡功能和安全问题，go 命令默认只使用 git 和 hg 从公共服务器下载代码。它将使用任何已知的版本控制系统从私有服务器下载与 GOPRIVATE 环境变量匹配的托管包的代码。仅允许 Git 和 Mercurial 的基本原理是，作为不受信任服务器的版本控制系统的客户端，它们是最受关注的。相比之下，Bazaar、Fossil 和 Subversion 主要用于受信任的、经过身份验证的环境，并且不会在攻击方面受到严格审查。

对版本控制命令的限制仅适用于直接使用版本控制系统下载代码时。从代理下载模块时，go 命令使用 GOPROXY 协议，这始终是被允许的。默认情况下，go 命令对公共模块使用 Go 模块镜像(proxy.golang.org)，并且仅在私有模块或镜像拒绝为公共包提供服务时（通常出于法律原因）回退去使用版本控制系统。因此，默认情况下，客户端仍然可以访问从 Bazaar、Fossil 或 Subversion 存储库提供的公共代码，因为这些下载使用 Go 模块镜像，承担了使用自定义沙箱运行版本控制命令的安全风险。

GOVCS 环境变量可用于更改允许下载特定模块的版本控制系统。GOVCS 环境变量在模块感知模式或 GOPATH 模式下构建包时用到。使用模块感知模式时，去匹配模块路径。使用 GOPATH 模式时，去匹配版本控制存储库根目录对应的导入路径。

GOVCS 环境变量的一般形式是逗号分隔的 pattern:vcslist 规则列表。该模式是一种全局模式，必须匹配模块或导入路径的一个或多个前导元素。vcslist 是允许的版本控制命令的管道分隔列表，或者 all 允许使用任何已知命令，或者 off 不允许任何内容。请注意，即使一个模块与 vcslist 的 off 模式匹配，如果源服务器使用 mod 方案，它仍然可能被下载，因为该方案指示 go 命令使用 GOPROXY 协

议来下载模块。先使用列表中最早的匹配模式，即使后面的模式也可能匹配。

例如，考虑：
GOVCS=github.com:git,evil.com:off,*:git|hg

使用此设置，以 github.com/开头的模块或导入路径的代码只能使用 git；evil.com 的路径不能使用任何版本控制命令，所有其他路径（*匹配所有内容）只能使用 git 或 hg。

特殊模式 public 和 private 匹配公共和私有模块或导入路径。如果路径与 GOPRIVATE 环境变量匹配，则该路径是私有的；否则它是公有的。

如果 GOVCS 环境变量中没有规则与特定模块或导入路径匹配，则 go 命令使用默认规则，现在可以用 GOVCS 表示法将其概括为 public:git|hg,private:all。

要允许对任何软件包不受限制地使用任何版本控制系统，请使用：
GOVCS=*:all

要禁用所有版本控制系统，请使用：
GOVCS=*:off

go env -w 命令可用于设置 GOVCS 环境变量。

GOVCS 环境变量是在 Go 1.16 中引入的。Go 的早期版本可以对任何模块使用任何已知的版本控制工具。

模块的 zip 文件(Module zip files)

模块版本以.zip 文件的形式分发。很少需要直接与这些文件交互，因为 go 命令会自动从模块代理和版本控制存储库中创建、下载和提取它们。但是，了解这些文件对于了解跨平台兼容性限制或在实现模块代理时仍然很有用。

go mod download 命令下载一个或多个模块的 zip 文件，然后将这些文件提取到模块缓存中。根据 GOPROXY 和其他环境变量，go 命令可以从代理下载 zip 文件或克隆源代码控制存储库并从中创建 zip 文件。-json 标志可用于在模块缓存中查找下载 zip 文件及其提取内容的位置。

golang.org/x/mod/zip 包可用于以编程方式创建、提取或检查 zip 文件的内容。

文件路径和大小的约束(File path and size constraints)

模块 zip 文件的内容有许多限制。这些约束确保 zip 文件可以在各种平台上安全

且一致地提取。

- 一个模块 zip 文件的大小最多为 500 MiB。其文件的总未压缩大小也限制为 500 MiB。go.mod 文件限制为 16 MiB。LICENSE 文件也限制为 16 MiB。存在这些限制是为了减轻对用户、代理和模块生态系统其他部分的拒绝服务攻击。在模块目录树中包含超过 500 MiB 文件的存储库应在 commit 时给模块版本加标签，仅包含构建模块包所需的文件；构建通常不需要视频、模型和其他大型的资源文件。
- 模块 zip 文件中的每个文件都必须以前缀$module@$version/开头，其中 $module 是模块路径，$version 是版本，例如 golang.org/x/mod@v0.3.0/。模块路径必须有效，版本必须有效且规范，版本必须与模块路径的主版本号后缀匹配。有关特定定义和限制，请参阅模块路径和版本号(Module paths and versions)。
- 文件模式、时间戳和其他元数据被忽略。
- 空目录（路径以斜杠结尾的条目）可以包含在模块 zip 文件中，但不会被提取。go 命令在它创建的 zip 文件中不包含空目录。
- 创建 zip 文件时会忽略符号链接和其他不规则文件，因为它们不能跨操作系统和文件系统移植，并且没有可移植的方式以 zip 文件格式表示它们。
- 创建 zip 文件时，会忽略名为 vendor 的目录中的文件，因为从不使用主模块之外的 vendor 目录。
- 包含 go.mod 文件的目录中的文件（模块根目录除外）在创建 zip 文件时会被忽略，因为它们不是模块的一部分。go 命令在解压 zip 文件时会忽略包含 go.mod 文件的子目录。
- 在 Unicode 大小写折叠下，一个 zip 文件中的任何两个文件都不能具有相等的路径（请参阅 strings.EqualFold）。这确保了可以在不区分大小写的文件系统上提取 zip 文件而不会发生冲突。
- go.mod 文件可能会或可能不会出现在顶级目录（$module@$version/go.mod）中。如果存在，它必须具有名称 go.mod（全部小写）。任何其他目录都不允许名为 go.mod 的文件。
- 模块中的文件和目录名称可能包含 Unicode 字母、ASCII 数字、ASCII 空格字符(U+0020)和 ASCII 标点字符!#$%&()+,-.=@[]^_{} ～。请注意，包路径可能不包含所有这些字符。有关差异，请参见 module.CheckFilePath 和 module.CheckImportPath。
- 直到第一个点的文件或目录名称不能是 Windows 上的保留文件名，无论大小写如何（CON、com1、NuL 等）。

私有模块(Private modules)

Go 模块经常在公共互联网不可及的版本控制服务器和模块代理上开发和分发。go 命令可以从私有源下载和构建模块，尽管它通常需要一些配置。

下面的环境变量可用于配置对私有模块的访问。有关详细信息，请参阅环境变量。有关控制发送到公共服务器的信息，另请参阅隐私(Privacy)。

- GOPROXY — 模块代理 URL 列表。go 命令将尝试按顺序从每个服务器下载模块。关键字 direct 指示 go 命令从开发模块的版本控制存储库下载模块，而不是使用代理。
- GOPRIVATE — 应该被视为私有的模块的路径前缀的全局匹配模式列表。充当 GONOPROXY 和 GONOSUMDB 的默认值。
- GONOPROXY — 不应该从代理下载的模块的路径前缀的全局匹配模式列表。go 命令将从开发它们的版本控制存储库中下载匹配的模块，而不考虑使用 GOPROXY。
- GONOSUMDB — 不应该使用公共校验值数据库 sum.golang.org 检查的模块的路径前缀的全局匹配模式列表。
- GOINSECURE — 可以通过 HTTP 和其他不安全协议检索的模块的路径前缀的全局匹配模式列表。

这些环境变量可以在开发环境中设置（例如，在.profile 文件中），也可以使用 go env -w 永久设置。

本节的剩余部分描述了对私有模块代理和版本控制存储库的访问的常见模式。

为所有模块提供服务的私人代理(Private proxy serving all modules)

为所有模块（公共的和私有的）提供服务的中央私有代理服务器为管理员提供了最大的控制权，并且为单个开发人员提供了最少的配置。

要将 go 命令配置为使用这样的服务器，请设置以下环境变量，将 https://proxy.corp.example.com 替换为你的代理 URL，将 corp.example.com 替换为你的模块前缀：
GOPROXY=https://proxy.corp.example.com
GONOSUMDB=corp.example.com

GOPROXY 的设置指示 go 命令仅从 https://proxy.corp.example.com 下载模块；go 命令不会连接到其他代理或版本控制存储库。

GONOSUMDB 的设置指示 go 命令不要使用公共校验值数据库来验证路径以 corp.example.com 开头的模块。

在此配置中运行的代理可能需要对私有版本控制服务器的读取访问权限。它还需要访问公共互联网以下载公共的模块的新版本。

有几种现有的 GOPROXY 服务器实现可以以这种方式使用。一个最小的实现是从模块缓存目录中提供文件，并使用 go mod download（具有合适的配置）来检索丢失的模块。

为私有模块提供服务的私人代理(Private proxy serving private modules)

私有代理服务器可以为私有模块提供服务，但同时也为公共可用模块提供服务。go 命令可以被配置为，当模块在私有代理服务器上不可用时，回退到模块的公共源。

要将 go 命令配置为以这种方式工作，请设置以下环境变量，将 https://proxy.corp.example.com 替换为代理 URL，将 corp.example.com 替换为模块前缀：

```
GOPROXY=https://proxy.corp.example.com,https://proxy.golang.org,direct
GONOSUMDB=corp.example.com
```

GOPROXY 的设置指示 go 命令首先尝试从 https://proxy.corp.example.com 下载模块。如果该服务器以 404（未找到）或 410（已消失）响应，go 命令将回退去使用 https://proxy.golang.org，然后直接连接到存储库。

GONOSUMDB 设置指示 go 命令不要使用公共校验值数据库来验证路径以 corp.example.com 开头的模块。

请注意，此配置中使用的代理仍可能控制对公共模块的访问，即使该代理不为它们提供服务。如果代理以 404 或 410 以外的错误状态码响应请求，则 go 命令将不会回退到 GOPROXY 列表中的后续条目。例如，对于具有不合适的许可证或具有已知安全漏洞的模块，代理可以响应 403（禁止）。

直接访问私有模块(Direct access to private modules)

go 命令可以配置为绕过公共代理并直接从版本控制服务器下载私有模块。当不能运行私有代理服务器时，这很有用。

要将 go 命令配置为以这种方式工作，请设置 GOPRIVATE 环境变量，将 corp.example.com 替换为私有模块的前缀：

```
GOPRIVATE=corp.example.com
```

在这种情况下不需要更改 GOPROXY 环境变量。它默认为 https://proxy.golang.org,direct，它指示 go 命令首先尝试从 https://proxy.golang.org 下载模块，然后如果该代理以 404（未找到）或 410（已消失）响应，则回退到直接连接（译者注：直接去版本控制服务器下载模块）。

GOPRIVATE 的设置指示 go 命令，对于以 corp.example.com 开头的模块，不要连接到代理或校验值数据库。

可能仍需要内部 HTTP 服务器来从模块路径解析存储库 URL。例如 go 命令在下载模块 corp.example.com/mod 时，会向 https://corp.example.com/mod?go-get=1 发送 GET 请求，并在响应中查找存储库的 URL。为避免此请求，请确保每个私有

模块路径都有一个 VCS 后缀（例如.git）标记存储库根前缀。例如，当 go 命令下载模块 corp.example.com/repo.git/mod 时，它将克隆位于 https://corp.example.com/repo.git 或 ssh://corp.example.com/repo.git 的 Git 存储库，无需额外的请求。

开发人员需要包含私有模块的存储库的读取权限。这可以在全局 VCS 配置文件（例如.gitconfig）中进行配置。最好将 VCS 工具配置为不需要交互式身份验证提示。默认情况下，调用 Git 时，go 命令通过设置 GIT_TERMINAL_PROMPT=0 来禁用交互式提示，但它也会遵循用户显式的设置（如果有的话）。

传递凭证给私有代理(Passing credentials to private proxies)

go 命令在与代理服务器通信时支持 HTTP 基本身份验证。

可以在.netrc 文件中指定凭证。例如，包含以下行的.netrc 文件将配置 go 命令以使用给定的用户名和密码连接到机器 proxy.corp.example.com。

```
machine proxy.corp.example.com
login jrgopher
password hunter2
```

可以使用 NETRC 环境变量设置文件的位置。如果未设置 NETRC，go 命令将在类 UNIX 平台上读取 $HOME/.netrc 或在 Windows 上读取%USERPROFILE%_netrc。

.netrc 中的字段用空格、制表符和换行符分隔。不幸的是，这些字符不能用于用户名或密码。另请注意，机器名称不能是完整的 URL，因此无法为同一机器上的不同路径指定不同的用户名和密码。

或者，可以直接在 GOPROXY 的 URL 中指定凭证。例如：

```
GOPROXY=https://jrgopher:hunter2@proxy.corp.example.com
```

采用这种方法时要小心：环境变量可能会出现在 shell 历史记录和日志中。

传递凭证给私有仓库(Passing credentials to private repositories)

go 命令可以直接从版本控制存储库下载模块。如果不使用私人代理，这对于私有模块是有必要的。请参阅直接访问私有模块(Direct access to private modules)进行配置。

go 命令在直接下载模块时会运行 git 等版本控制工具。这些工具执行自己的身份认证机制，因此你可能需要在特定于工具的配置文件（例如.gitconfig）中配置凭据。

为确保此操作顺利进行，请确保 go 命令使用正确的存储库 URL，并且版本控制工具不需要以交互方式输入密码。go 命令更喜欢 https://URLs 而不是 ssh://等其他方案，除非在查找存储库 URL 时指定了方案。特别是对于 GitHub 存储库，go 命令假定为 https://。

对于大多数服务器，你可以将客户端配置为通过 HTTP 进行身份认证。例如，GitHub 支持使用 OAuth 个人访问令牌作为 HTTP 密码。你可以将 HTTP 密码存储在.netrc 文件中，就像传递凭据给私有代理(passing credentials to private proxies)一样。

或者，你可以将 https://方案的 URL 重写为另一个方案。例如，在.gitconfig 中：
[url "git@github.com:"]
    insteadOf = https://github.com/

有关更多信息，请参阅为什么“go get”在克隆存储库时使用 HTTPS？

隐私(Privacy)

go 命令可以从模块代理服务器和版本控制系统下载模块和元数据。环境变量 GOPROXY 控制使用哪些代理服务器。环境变量 GOPRIVATE 和 GONOPROXY 控制从代理获取哪些模块。

GOPROXY 的默认值为：
https://proxy.golang.org,direct

有了这个设置，当 go 命令下载一个模块或模块元数据时，它会首先向 proxy.golang.org 发送请求，这是一个由 Google 运营的公共模块代理（隐私策略）。有关在每个请求中发送哪些信息的详细信息，请参阅 GOPROXY 协议(GOPROXY protocol)。go 命令不会传输个人身份信息，但会传输所请求的完整模块路径。如果代理以 404（未找到）或 410（已消失）状态响应，go 命令将尝试直接连接到提供模块的版本控制系统。有关详细信息，请参阅版本控制系统(Version control systems)。

GOPRIVATE 或 GONOPROXY 环境变量可以设置为匹配模块前缀的全局模式列表，这些模块是私有的，不应从任何代理请求。例如：
GOPRIVATE=*.corp.example.com,*.research.example.com

GOPRIVATE 只是作为 GONOPROXY 和 GONOSUMDB 的默认值，因此没有必要设置 GONOPROXY，除非 GONOSUMDB 应该有不同的值。当模块路径与 GONOPROXY 匹配时，go 命令会忽略该模块的 GOPROXY 并直接从其版本控制存储库中获取它。当没有代理为私有模块提供服务时，这很有用。请参阅直接访问私有模块(Direct access to private modules)。

如果存在为所有模块提供服务的可信代理，则不应设置 GONOPROXY。例如，

如果 GOPROXY 设置为一个源，则 go 命令将不会从其他源下载模块。在这种情况下仍应设置 GONOSUMDB。
GOPROXY=https://proxy.corp.example.com
GONOSUMDB=*.corp.example.com,*.research.example.com

如果有一个受信任的代理只为私有模块提供服务，则不应设置 GONOPROXY，但必须注意确保代理以正确的状态代码进行响应。例如，考虑以下配置：
GOPROXY=https://proxy.corp.example.com,https://proxy.golang.org
GONOSUMDB=*.corp.example.com,*.research.example.com

假设由于拼写错误，开发人员试图下载一个不存在的模块。
go mod download corp.example.com/secret-product/typo@latest

go 命令首先从 proxy.corp.example.com 请求这个模块。如果该代理以 404（未找到）或 410（已消失）响应，则 go 命令将回退到 proxy.golang.org，并在请求 URL 中传输 secret-product 路径。如果私有代理响应任何其他错误代码，go 命令会打印错误信息并且不会回退到其他源。

除了代理之外，go 命令还可以连接到校验值数据库，以验证 go.sum 中未列出的模块的加密散列值。GOSUMDB 环境变量设置校验值数据库的名称、URL 和公钥。GOSUMDB 的默认值为 sum.golang.org，由 Google 运营的公共校验值数据库（隐私政策）。有关每个请求传输的内容的详细信息，请参阅校验值数据库(Checksum database)。与代理一样，go 命令不会传输个人身份信息，但会传输所请求的完整模块路径，并且校验值数据库无法计算非公有模块的校验值。

GONOSUMDB 环境变量可以设置为指示哪些模块是私有的并且不应从校验值数据库请求。GOPRIVATE 作为 GONOSUMDB 和 GONOPROXY 的默认值，因此没有必要设置 GONOSUMDB，除非 GONOPROXY 应该具有不同的值。

代理可以设置校验值数据库镜像。如果 GOPROXY 中的代理执行此操作，则 go 命令将不会直接连接到校验值数据库。

GOSUMDB 可以设置为 off 以完全禁用校验值数据库。使用此设置，go 命令将不会验证下载的模块，除非它们已经在 go.sum 中。请参阅验证模块(Authenticating modules)。

模块缓存(Module cache)

模块缓存(module cache)是 go 命令存储下载的模块文件的目录。模块缓存不同于构建缓存，后者包含已编译的包和其他构建文件。

模块缓存的默认位置是$GOPATH/pkg/mod。要使用不同的位置，请设置 GOMODCACHE 环境变量。

模块缓存没有最大大小限制，go 命令不会自动删除其内容。

模块缓存可能由在同一台机器上开发的多个 Go 项目共享。无论主模块的位置在哪里，go 命令都将使用相同的缓存。go 命令的多个实例可以同时安全地访问同一个模块缓存。

go 命令在缓存中创建具有只读权限的模块源文件和目录，以防止在下载模块后对模块进行意外更改。这有一个令人遗憾的副作用，即使用 rm -rf 之类的命令难以删除缓存。可以使用 go clean -modcache 删除缓存。或者，当使用-modcacherw 标志时，go 命令将创建具有读写权限的新目录。这增加了编辑器、测试和其他程序修改模块缓存中文件的风险。go mod verify 命令可用于检测主模块依赖项是否有修改。它扫描每个模块依赖项的提取内容，并确认它们是否与 go.sum 中的预期散列值相匹配。

下表解释了模块缓存中大多数文件的用途。省略了一些临时文件（锁定文件、临时目录）。对于每个路径，$module 是一个模块路径，$version 是一个版本号。以斜杠(/)结尾的路径是目录。模块路径和版本号中的大写字母使用感叹号进行转义（例如 Azure 转义为!azure）以避免在不区分大小写的文件系统上发生冲突。

路径	描述
$module@$version/	包含模块.zip 文件的提取内容的目录。这用作下载模块的模块根目录。如果原始模块没有 go.mod 文件，它将不包含 go.mod 文件。
cache/download/	此目录包含从模块代理下载的文件和来自版本控制系统的文件。此目录的布局遵循 GOPROXY 协议，因此当由 HTTP 文件服务器提供服务或使用 file:// URL 引用时，此目录可以用作代理。
cache/download/$module/@v/list	已知版本列表（参见 GOPROXY 协议）。这可能会随着时间而改变，因此 go 命令通常会获取一个新副本，而不是重新使用该文件。
cache/download/$module/@v/$version.info	有关版本的 JSON 元数据。（参见 GOPROXY 协议）。这可能会随着时间而改变，因此 go 命令通常会获取一个新副本，而不是重新使用该文件。
cache/download/$module/@v/$version.mod	此版本的 go.mod 文件（参见 GOPROXY 协议）。如果原始模块没有 go.mod 文件，这是一个合成的、不包含依赖项的文件。

cache/download/$module/@v/$version.zip	模块的压缩内容（参见 GOPROXY 协议和模块 zip 文件 (Module zip files)）。
cache/download/$module/@v/$version.ziphash	.zip 文件中文件的加密哈希值。注意，.zip 文件本身没有被计算散列值，因此文件顺序、压缩、对齐和元数据不会影响该加密哈希值。使用模块时，go 命令会验证此哈希值是否与 go.sum 中的相应行匹配。go mod verify 命令检查模块.zip 文件和提取目录的哈希值是否与这些文件匹配。
cache/download/sumdb/	此目录包含从校验值数据库（通常是 sum.golang.org）下载的文件。
cache/vcs/	此目录包含直接从其版本控制存储库源中获取的模块的克隆。目录名称是从存储库类型和 URL 派生的十六进制编码哈希值。存储库针对磁盘大小进行了优化。例如，克隆的 Git 存储库在可能的情况下是空的或浅层的。

验证模块(Authenticating modules)

当 go 命令将模块 zip 文件或 go.mod 文件下载到模块缓存中时，它会计算加密哈希值并将其与已知值进行比较，以验证文件自首次下载以来没有更改。如果下载的文件没有正确的哈希值，go 命令会报告安全错误。

对于 go.mod 文件，go 命令根据文件内容计算哈希值。对于模块 zip 文件，go 命令以确定的顺序从存档中文件的名称和内容计算哈希值。哈希值不受文件顺序、压缩、对齐和其他元数据的影响。有关哈希值实现的详细信息，请参见 golang.org/x/mod/sumdb/dirhash。

go 命令将每个哈希值与主模块的 go.sum 文件中的相应行进行比较。如果哈希值与 go.sum 中的哈希值不同，go 命令会报安全错误并删除下载的文件，而不会将其添加到模块缓存中。

如果 go.sum 文件不存在，或者它不包含下载文件的哈希值，则 go 命令可以使用校验值数据库来验证哈希值，该数据库是公开可用模块的全局哈希值源。验证哈希值后，go 命令将其添加到 go.sum 并将下载的文件添加到模块缓存中。如果模块是私有的（由 GOPRIVATE 或 GONOSUMDB 环境变量匹配）或校验值数据库

被禁用（通过设置 GOSUMDB=off），go 命令接受哈希值并将文件添加到模块缓存而不验证它。

模块缓存通常由系统上的所有 Go 项目共享，每个模块可能有自己的 go.sum 文件，其哈希值可能不同。为了避免信任其他模块，go 命令在访问模块缓存中的文件时使用主模块的 go.sum 验证哈希值。Zip 文件哈希值的计算成本很高，因此 go 命令检查与 zip 文件一起存储的预先计算好的哈希值，而不是重新计算。go mod verify 命令可用于检查 zip 文件和提取的目录在添加到模块缓存后是否未被修改。

go.sum 文件(go.sum files)

一个模块的根目录中可能有一个名为 go.sum 的文本文件，旁边是它的 go.mod 文件。go.sum 文件包含模块直接和间接依赖项的加密哈希值。当 go 命令将模块.mod 或.zip 文件下载到模块缓存中时，它会计算一个哈希值并检查该哈希值是否与主模块的 go.sum 文件中的相应哈希值匹配。如果模块没有依赖项或者所有依赖项都使用 replace 指令替换为本地目录，则 go.sum 可能为空或不存在。

go.sum 中的每一行都有三个由空格分隔的字段：模块路径、版本号（可能以 /go.mod 结尾）和哈希值。

- 模块路径是哈希值所属模块的名称。
- 版本号是哈希值所属模块的版本。如果版本号以/go.mod 结尾，则哈希值仅用于模块的 go.mod 文件；否则，哈希值用于模块的.zip 文件中的文件。
- 哈希值那列由算法名称（例如 h1）和 base64 编码的加密哈希值组成，以冒号(:)分隔。目前，SHA-256(h1)是唯一支持的哈希算法。如果将来发现 SHA-256 中的漏洞，将添加对另一种算法（名为 h2 等）的支持。

go.sum 文件可能包含一个模块的多个版本的哈希值。go 命令可能需要从依赖项的多个版本加载 go.mod 文件，以便执行最小版本选择算法。go.sum 还可能包含不再需要的模块版本的哈希值（例如，在升级之后）。go mod tidy 将添加缺失的哈希值，并从 go.sum 中删除不必要的哈希值。

校验值数据库(Checksum database)

校验值数据库是 go.sum 行的全局来源。go 命令可以在许多情况下使用它来检测代理或原始服务器的不当行为。

校验值数据库让所有公开可用的模块版本获得全局一致性和可靠性。它使不受信任的代理成为可能，因为它们无法提供错误的代码而不会被警告。它还确保与特定版本相关联的位不会从一天到另一天发生变化，即使模块的作者随后更改了其在存储库中的标签。

校验值数据库由谷歌运行的 sum.golang.org 提供服务。它是由 Trillian 支持的 go.sum 行哈希值的透明日志（或“Merkle 树”）。Merkle 树的主要优点是独立审

计员可以验证它没有被篡改，因此它比简单的数据库更值得信赖。

go 命令使用最初在 Proposal: Secure the Public Go Module Ecosystem 中概述的协议与校验值数据库交互。

下表指定了校验值数据库必须响应的查询。对于每个路径，$base 是校验值数据库 URL 的路径部分，$module 是模块路径，$version 是版本号。例如，如果校验值数据库 URL 是 https://sum.golang.org，并且客户端正在请求版本号为 v0.3.2 的模块 golang.org/x/text 的记录，则客户端将发送 GET 请求 https://sum.golang.org/lookup/golang.org/x/text@v0.3.2。

为了避免在不区分大小写的文件系统中提供服务时产生歧义，$module 和$version 元素通过将每个大写字母替换为感叹号后跟相应的小写字母来进行大小写编码。这允许模块 example.com/M 和 example.com/m 都存储在磁盘上，因为前者被编码为 example.com/!m。

被方括号包围的部分路径，例如[.p/$W]表示可选值。

路径	描述
$base/latest	返回最新日志的签名、编码树的描述。此签名描述采用便笺的形式，它是由一个或多个服务器密钥签名的文本，可以使用服务器的公钥进行验证。树描述提供树的大小和该大小的树头的散列。这种编码在 golang.org/x/mod/sumdb/tlog#FormatTree 中有描述。
$base/lookup/$module@$version	返回有关$version 的$module 条目的日志记录编号，后跟记录的数据（即$version 的$module 的 go.sum 行）和包含该记录的签名、编码树的描述。
$base/tile/$H/$L/$K[.p/$W]	返回一个[log tile](https://research.swtch.com/tlog#serving_tiles)，它是一组组成部分日志的哈希值。每个图块在图块级别$L 的二维坐标中定义，从左侧开始第$K 个，图块高度为$H。可选的.p/$W 后缀表示只有$W 哈希值的部分日志图块。如果未找到部分图块，客户端必须回退去获取完整图块。
$base/tile/$H/data/$K[.p/$W]	返回/tile/$H/0/$K[.p/$W]中叶哈希值的记录数据（带有字面上的 data 路径元素）。

如果 go 命令查询校验值数据库，那么第一步是通过/lookup 端点检索记录数据。如果模块版本尚未记录在日志中，校验值数据库将在回复之前尝试从源服务器获取它。这个/lookup 数据提供了这个模块版本的校验值以及它在日志中的位置，它通知客户端应该获取哪些图块来执行证明。go 命令在将新的 go.sum 行添加到主模块的 go.sum 文件之前执行“包含”证明（特定记录存在于日志中）和“一

致性”证明（编码树没有被篡改）。重要的是，如果没有先根据已签名的树的哈希值和客户端的已签名的树的哈希值的时间轴对其进行身份验证，则绝不应使用来自/lookup 的数据。

校验值数据库提供的签名树哈希值和新切片存储在模块缓存中，因此 go 命令只需获取丢失的切片。

go 命令不需要直接连接到校验值数据库。它可以通过校验值数据库的镜像并支持上述协议的模块代理请求模块的校验值。这对于禁止组织外部请求的私人、公司代理特别有用。

GOSUMDB 环境变量标识要使用的校验值数据库的名称以及可选的公钥和 URL，如下所示：
GOSUMDB="sum.golang.org"
GOSUMDB="sum.golang.org+<publickey>"
GOSUMDB="sum.golang.org+<publickey> https://sum.golang.org"

go 命令已知 sum.golang.org 的公钥，并且已知 sum.golang.google.cn 这个名字（在中国大陆可用）连接到 sum.golang.org 校验值数据库；使用任何其他数据库都需要明确提供公钥。URL 默认为 https://，后跟数据库名称。

GOSUMDB 默认为 sum.golang.org，是由 Google 运行的 Go 校验值数据库。有关服务的隐私政策，请参阅 https://sum.golang.org/privacy。

如果 GOSUMDB 设置为 off，或者如果使用-insecure 标志调用 go get，则不会查询校验值数据库，并且接受所有无法识别的模块，代价是放弃所有模块的已验证可重复下载的安全保证。绕过特定模块的校验值数据库的更好方法是使用 GOPRIVATE 或 GONOSUMDB 环境变量。有关详细信息，请参阅私有模块(Private Modules)。

go env -w 命令可用于为将来的 go 命令调用设置这些环境变量。

环境变量(Environment variables)

go 命令中的模块行为可以使用下面列出的环境变量进行配置。此列表仅包括与模块相关的环境变量。请参阅 go help environment 以获取 go 命令识别的所有环境变量的列表。

环境变量	描述
GO111MODULE	控制 go 命令是在模块感知模式还是 GOPATH 模式下运行。三个值被认可： ● off：go 命令忽略 go.mod 文件并以 GOPATH 模式运行。 ● on（或未设置）：go 命令在模块感知模式下运行，即使没有 go.mod 文件存在。

	• auto：如果当前目录或任何父目录中存在 go.mod 文件，则 go 命令以模块感知模式运行。在 Go 1.15 及更低版本中，这是默认设置。 有关更多信息，请参阅模块感知命令(Module-aware commands)。
GOMODCACHE	go 命令存储下载的模块和相关文件的目录。有关此目录结构的详细信息，请参阅模块缓存(Module cache)。 如果未设置 GOMODCACHE，则默认为 $GOPATH/pkg/mod。
GOINSECURE	以逗号分隔的模块路径前缀的全局匹配模式的列表（在 Go 的 path.Match 的语法中）可能总是以不安全的方式获取模块。仅适用于直接获取的依赖项。 与 go get 上的-insecure 标志不同，GOINSECURE 不会禁用模块校验值数据库验证。GOPRIVATE 或 GONOSUMDB 可用于实现此目的。
GONOPROXY	以逗号分隔的模块路径前缀的全局匹配模式（在 Go 的 path.Match 的语法中）的列表，应始终直接从版本控制存储库中获取模块，而不是从模块代理中获取。 如果未设置 GONOPROXY，则默认使用 GOPRIVATE。请参阅隐私(Privacy)。
GONOSUMDB	以逗号分隔的模块路径前缀的全局匹配模式（在 Go 的 path.Match 的语法中）的列表，go 不应该使用校验值数据库验证这些模块的校验值。 如果未设置 GONOSUMDB，则默认使用 GOPRIVATE。请参阅隐私(Privacy)。
GOPATH	在 GOPATH 模式下，GOPATH 环境变量是可能包含 Go 代码的目录的列表。 在模块感知模式下，模块缓存存储在第一个 GOPATH 目录的 pkg/mod 子目录中。缓存之外的模块源代码可以存储在任何目录中。 如果 GOPATH 未设置，则默认使用户主目录下的 go 子目录。
GOPRIVATE	以逗号分隔的应该被视为私有的模块的路径前缀的全局匹配模式(在 Go 的 path.Match 的语法中)的列表。GOPRIVATE 是 GONOPROXY 和 GONOSUMDB 的默认值。请参阅隐私(Privacy)。GOPRIVATE 还确定模块对于 GOVCS 来说是否被认为是私有的。

	模块代理 URL 的列表，以逗号(,)或管道符号(|)分隔。当 go 命令查找有关模块的信息时，它会依次联系列表中的每个代理，直到收到成功的响应或终端错误。代理可能会以 404（未找到）或 410（已消失）状态进行响应，以指示该模块在该服务器上不可用。 go 命令的错误回退行为由 URL 之间的分隔符决定。如果代理 URL 后跟逗号，则 go 命令在 404 或 410 错误后回退到下一个 URL；所有其他错误都被认为是终端错误。如果代理 URL 后跟管道符号，则 go 命令会在任何错误（包括超时等非 HTTP 错误）后回退到下一个源。
GOPROXY	GOPROXY 的 URL 可能有 https、http 或 file 的方案。如果 URL 没有包含方案，则假定为 https。模块缓存可以直接用作文件代理： GOPROXY=file://$(go env GOMODCACHE)/cache/download 可以使用两个关键字代替代理的 URL： ● off：禁止从任何来源下载模块。 ● direct：直接从版本控制存储库下载，而不是使用模块代理。 GOPROXY 默认为 https://proxy.golang.org,direct。在该配置下，go 命令首先联系 Google 运行的 Go 模块镜像，如果镜像里没有该模块，则回退到直接连接。有关镜像的隐私政策，请参阅 https://proxy.golang.org/privacy。可以设置 GOPRIVATE 和 GONOPROXY 环境变量以防止使用代理下载特定模块。有关私有代理配置的信息，请参阅隐私(Privacy)。 有关如何使用代理的更多信息，请参阅模块代理(Module proxies)和将包解析为模块(Resolving a package to a module)。
GOSUMDB	标识要使用的校验值数据库的名称以及可选的公钥和 URL。例如： GOSUMDB="sum.golang.org" GOSUMDB="sum.golang.org+<publickey>" GOSUMDB="sum.golang.org+<publickey> https://sum.golang.org" go 命令已知 sum.golang.org 的公钥，并且已知 sum.golang.google.cn（在中国大陆可用）连接到 sum.golang.org 数据库；使用任何其他数据库都需要明确提供公钥。URL 默认为 https://后跟数据库名称。

	GOSUMDB 默认为 sum.golang.org，是由 Google 运营的 Go 校验值数据库。有关服务的隐私政策，请参阅 https://sum.golang.org/privacy。 如果 GOSUMDB 设置为 off 或者使用-insecure 标志调用 go get，则不会查询校验值数据库，并且接受所有无法识别的模块，代价是放弃所有模块的已验证的可重复下载的安全保证。绕过特定模块的校验值数据库的更好方法是使用 GOPRIVATE 或 GONOSUMDB 环境变量。 有关更多信息，请参阅认证模块(Authenticating modules)和隐私(Privacy)。
GOVCS	控制 go 命令可用于下载公有和私有模块（由它们的路径是否与 GOPRIVATE 中的模式匹配）或与全局模式匹配的其他模块的版本控制工具集。 如果未设置 GOVCS，或者如果模块与 GOVCS 中的任何模式都不匹配，则 go 命令可以将 git 和 hg 用于公共模块，或任何已知的版本控制工具用于私有模块。具体来说，go 命令的行为就像 GOVCS 设置为： public:git|hg,private:all 有关完整说明，请参阅使用 GOVCS 控制版本控制工具(Controlling version control tools with GOVCS)。
GOWORK	`GOWORK`环境变量指示`go`命令进入工作区模式，使用提供的[`go.work`文件](#go-work-file)来定义工作区。如果`GOWORK`设置为`off`，则工作区模式被禁用。这可用于在单模块模式下运行`go`命令：例如，`GOWORK=off go build .`以单模块模式构建`.`包。如果`GOWORK`为空，则`go`命令将搜索`go.work`文件，如[Workspaces](#workspaces)部分所述。

词汇表(Glossary)

构建约束(build constraint)：一个条件，用于确定在编译包时是否使用 Go 源文件。构建约束可以用文件名后缀（例如，foo_linux_amd64.go）或构建约束注释（例如，// +build linux,amd64）来表示。请参阅构建约束(Build Constraints)。
构建列表(build list)：将用于构建命令的模块版本的列表，例如 go build、go list 或 go test。构建列表由主模块的 go.mod 文件和依赖项模块中的 go.mod 文件确定，使用最小版本选择算法。构建列表包含模块图中所有模块的版本，而不仅仅是与特定命令相关的版本。
规范版本号(canonical version)：格式正确的版本号，没有+incompatible 以外的构建元数据后缀。例如，v1.2.3 是规范版本号，但 v1.2.3+meta 不是。

当前模块(current module)：主模块的同义词。
弃用的模块(deprecated module)：其作者不再支持的模块（尽管出于此目的，主版本号不同的模块被视为不同的模块）。弃用的模块在其最新版本的 go.mod 文件中标有弃用注释。
直接依赖(direct dependency)：一个包，其路径出现在.go 源文件的 import 声明中，用于主模块中的包或测试，或包含此类包的模块。（比较间接依赖(indirect dependency)。）
直接模式(direct mode)：环境变量的设置，导致 go 命令直接从版本控制系统下载模块，而不是使用模块代理。GOPROXY=direct 对所有模块执行此操作。GOPRIVATE 和 GONOPROXY 对匹配模式列表的模块执行此操作。
go.mod 文件：定义模块路径、依赖项和其他元数据的文件。出现在模块的根目录中。请参阅 go.mod 文件部分。
go.work 文件：定义要在工作区中使用的模块集的文件。请参阅 go.work 文件部分。
导入路径(import path)：用于在 Go 源文件中导入包的字符 1 串。与包路径(package path)同义。
间接依赖(indirect dependency)：由主模块中的包或测试导入的包，但其路径未出现在主模块中的任何 import 声明中；或出现在模块图中但不提供任何由主模块直接导入的包的模块。（比较直接依赖(direct dependency)。）
模块懒加载(lazy module loading)：Go 1.17 中的一个更改，避免在指定 go 1.17 或更高版本的模块中为不需要模块图的命令加载模块图。请参阅模块懒加载(Lazy module loading)。
主模块(main module)：调用 go 命令的模块。主模块由当前目录或父目录中的 go.mod 文件定义。请参阅模块、包和版本号(Modules, packages, and versions)。
主版本号(major version)：语义版本号中的第一个数字（例如 v1.2.3 中的 1）。在具有不兼容更改的版本中，主版号本必须递增，次版本号和补丁版本号必须设置为 0。主版本号为 0 的版本被认为是不稳定的。
主版本号子目录(major version subdirectory)：版本控制存储库中的子目录，匹配模块的主版本号后缀，可以在其中定义模块。例如，根路径 example.com/mod 的存储库中的模块 example.com/mod/v2 可能定义在存储库根目录或主版本号子目录 v2 中。请参阅存储库中的模块目录(Module directories within a repository)。
主版本号后缀(major version suffix)：与主版本号匹配的模块路径后缀。例如，example.com/mod/v2 中的/v2。主版本号后缀在 v2.0.0 及更高版本中是必需的，在早期版本中不被允许。请参阅主版本号后缀(Major version suffixes)部分。
最小版本选择(minimal version selection (MVS))：用于确定将在构建中使用的所有模块的版本的算法。有关详细信息，请参阅最小版本选择(Minimal version selection)部分。
次版本号(minor version)：语义版本号中的第二个数字（例如 v1.2.3 中的 2）。在具有新的向后兼容功能的版本中，次版本号必须递增，并且补丁版本号必须设置为 0。
模块(module)：一起发布、版本号化和分发的包的集合。
模块缓存(module cache)：存储下载的模块的本地目录，位于 GOPATH/pkg/mod。请参阅模块缓存(Module cache)。
模块图(module graph)：模块依赖形成的有向图，根位于主模块。图中的每个顶

点都是一个模块；每条边都是 go.mod 文件中 require 语句里的一个版本（也受主模块的 go.mod 文件中的 replace 和 exclude 语句的影响）。

模块图化简(module graph pruning)：Go 1.17 中的一个更改，它通过省略指定 go 1.17 或更高版本的模块的传递的依赖关系来减小模块图的大小。请参阅模块图化简(Module graph pruning)。

模块路径(module path)：标识模块并充当模块内包导入路径前缀的路径。例如，“golang.org/x/net”。

模块代理(module proxy)：实现 GOPROXY 协议的 Web 服务。go 命令从模块代理下载版本信息、go.mod 文件和模块 zip 文件。

模块根目录(module root directory)：包含定义模块的 go.mod 文件的目录。

模块子目录(module subdirectory)：存储库根路径之后的模块路径部分，指示定义模块的子目录。当非空时，模块子目录也是语义版本号标签的前缀。即使模块在主版本号子目录中，模块子目录不包括主版本号后缀，如果存在的话。请参阅模块路径(Module paths)。

包(package)：同一目录中一起编译的源文件的集合。请参阅 Go 语言规范中的包部分(Packages section)。

包路径(package path)：唯一标识包的路径。包路径是模块路径再加上模块内的子目录的路径。例如“golang.org/x/net/html”是模块“golang.org/x/net”中“html”子目录里的包的包路径。是导入路径(import path)的同义词。

补丁版本号(patch version)：语义版本号中的第三个数字（v1.2.3 中的 3）。在未更改模块公共接口的新版本中，补丁版本号必须递增。

预发布版本号(pre-release version)：带有破折号的版本号，后跟一系列用点分隔的标识符，紧跟补丁版本号，例如 v1.2.3-beta4。预发布版本被认为是不稳定的，并且不假定与其他版本兼容。预发布版本号排在对应的发布版本号之前，例如 v1.2.3-pre 排在 v1.2.3 之前。另请参阅发布版本号(release version)。

伪版本号(pseudo-version)：对修订标识符（例如 Git commit 哈希）和来自版本控制系统的时间戳进行编码的版本号。例如，v0.0.0-20191109021931-daa7c04131f5。用于与非模块存储库的兼容性以及标记版本不可用的其他情况。

发布版本号(release version)：没有预发布后缀的版本号。例如，v1.2.3，而不是 v1.2.3-pre。另请参阅预发布版本号(pre-release version)。

存储库根路径(repository root path)：对应于版本控制存储库根目录的模块路径部分。请参阅模块路径(Module paths)。

撤回版本(retracted version)：一个不应该依赖的版本，要么是因为它过早发布，要么是因为它发布后发现了一个严重的问题。见撤回指令(retract directive)。

语义版本号标签(semantic version tag)：版本控制存储库中将版本号映射到特定修订版的标签。请参阅将版本映射到 commit(Mapping versions to commits)。

vendor 目录(vendor directory)：一个名为 vendor 的目录，其中包含来自其他模块的包，这些包在主模块中构建包时用到。由 go mod vendor 维护。请参阅 vendor 模式(Vendoring)。

版本号(version)：模块不可变快照的标识符，写为字母 v 后跟语义版本号。请参阅版本号(Versions)部分。

工作空间(workspace)：磁盘上的一组模块，在运行最小版本选择(MVS)算法时用作根模块。请参阅工作空间(Workspaces)部分